Міністерство освіти і науки України
Український державний університет науки і технологій

Навчально-науковий інститут
«Український державний хіміко-технологічний університет»
(назва навчально-наукового інституту)

Факультет комп'ютерних наук та інженерії
(повна назва факультету)

Кафедра комп'ютерно-інтегрованих технологій та робототехніки
(повна назва кафедри)

Пояснювальна записка

до дипломної роботи
бакалавр
(освітній рівень)

на тему: «Розробка розподіленої системи моніторингу мікроклімату приміщень на базі Raspberry 5 Pi та бездротових IoT-вузлів»

Виконав студент 4 курсу, групи КІ-22
спеціальності
123 «Комп'ютерна інженерія»
(код і назва спеціальності)

Хамаза М.Р. _____
(прізвище, ім'я, по-батькові) (підпис)

Керівник Дубовик Т.М.
(прізвище, ім'я, по-батькові) (підпис)

Рецензент Хорошилов С. В.
(прізвище, ім'я, по-батькові) (підпис)

Дніпро – 2026 рік

Український державний університет науки і технологій
(повне найменування вищого навчального закладу)

Навчально-науковий інститут
«Український державний хіміко-технологічний університет»
(назва навчально-наукового інституту)

Факультет, відділення_комп'ютерних наук та інженерії_____
Кафедра__комп'ютерно-інтегрованих технологій та робототехніки_____
Освітній рівень_бакалавр_____
Спеціальність _123 Комп'ютерна інженерія_____
(код і назва)
Спеціалізація _____
(шифр і назва)
Освітня програма _Комп'ютерна інженерія_____
(назва)

ЗАТВЕРДЖУЮ

В.о. завідувача кафедри _____
ЛЕВЧУК І.Л., канд. техн. наук, доц.
"____" _____ 2026 року

З А В Д А Н Н Я
НА ДИПЛОМНУ РОБОТУ СТУДЕНТУ

Хамаза Михайло Русланович
(прізвище, ім'я, по батькові)

1.Тема роботи: Розробка розподіленої системи моніторингу мікроклімату____ приміщень на базі Raspberry 5 Pi та бездротових IoT-вузлів_____ керівник роботи ст. викладач Дубовик Т.М. _____
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом вищого навчального закладу від 02.03.2026 р. №77-к___
2. Строк подання студентом роботи _08.06.2026 р._____
3. Вихідні дані до роботи: Матеріали, отримані під час виробничої практики, та відомості з технічної літератури._____
4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) Вступ. Загальний розділ. Основний розділ. Проектний розділ. Охорона праці та безпека в надзвичайних ситуаціях. Організаційно-економічний_____ розділ_____
5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень) _Презентація до дипломної роботи_____

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Охорона праці та безпека в надзвичайних ситуаціях	Дубовик Тетяна Миколаївна, ст. викл		
Організаційно-економічний розділ	Дубовик Тетяна Миколаївна, ст. викл		
Консультант	Романчук Олександр Олександрович, асистент		

7. Дата видачі завдання 14 лютого 2026 року

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проекту (роботи)	Строк виконання етапів роботи	Примітка
1	Бібліотечний огляд за темою	лютий	виконано
2	Робота над загальним розділом	лютий	виконано
3	Робота над основним розділом	березень-квітень	виконано
4	Робота над проектним розділом	березень-квітень	виконано
5	Розділ «Охорона праці та безпека в надзвичайних ситуаціях»	травень	виконано
6	Організаційно-економічний розділ	травень	виконано
7	Презентація	травень	виконано
8	Передзахист	25.05.2026	виконано
9	Подання дипломної роботи на підпис завідувачу кафедри	04.06 – 10.06.2026	виконано
10	Захист дипломної роботи	17.06.2026	виконано

Студент _____ Хамаза М. Р.
(підпис) (прізвище, ім'я, по-батькові)

Керівник роботи _____ Дубовик Т.М.
(підпис) (прізвище, ім'я, по-батькові)

РЕФЕРАТ

Дипломна робота містить 74 сторінки, 17 рисунків, 13 таблиць, 31 джерело.

Об'єкт розробки – розподілена система моніторингу мікроклімату приміщень.

Мета роботи – розробити розподілену комп'ютерну систему моніторингу мікроклімату приміщень на базі Raspberry Pi 5 та бездротових IoT-вузлів з відображенням даних у веб-інтерфейсі.

В загальному розділі дипломної роботи наведені теоретичні відомості про IoT-системи та їх класифікацію, огляд існуючих систем моніторингу мікроклімату, характеристики мікроконтролера ESP32 та одноплатного комп'ютера Raspberry Pi 5 як обчислювального вузла системи.

В основному розділі проведено порівняльний аналіз бездротових технологій передачі даних та обґрунтовано вибір протоколу HTTP/REST для взаємодії між вузлами системи.

В розрахунковій частині роботи представлена розробка спеціалізованої розподіленої системи моніторингу мікроклімату: схема збору даних із сенсора BME680 через шину I2C мікроконтролером ESP32, передача показань на сервер за протоколом HTTP, зберігання у базі даних PostgreSQL з використанням типу JSONB, розробка REST API на базі Spring Boot та реалізація веб-інтерфейсу для візуалізації часових рядів.

Актуальність розробки: результати даної розробки можуть мати позитивний вплив для виробничих цехів, житлових приміщень, серверних кімнат та медичних закладів – об'єктивний контроль температури, вологості, атмосферного тиску та якості повітря дозволяє своєчасно виявляти відхилення від норми, приймати обґрунтовані рішення щодо вентиляції та опалення, а також забезпечити захист обладнання і здоров'я людей.

Ключові слова: RASPBERRY PI 5, ESP32, IoT, МОНІТОРИНГ МІКРОКЛІМАТУ, BME680, REST API, SPRING BOOT, POSTGRESQL, HTTP, РОЗПОДІЛЕНА СИСТЕМА.

Зміст

ВСТУП

Забезпечення комфортного мікроклімату у приміщенні вже давно розглядається не лише як питання самопочуття. У виробничих цехах, серверних кімнатах, медичних закладах і навіть у житлових квартирах контроль температури, вологості та атмосферного тиску є обов'язковою вимогою, недотримання якої може призвести до матеріальних збитків або погіршення здоров'я. Промислові системи моніторингу залишаються дорогими та надмірно складними для невеликих об'єктів, тоді як недорогі автономні пристрої не забезпечують зберігання даних і аналізу їх у динаміці.

Протягом останнього десятиліття розвиток ринку одноплатних комп'ютерів і мікроконтролерів зробив розробку подібних систем доступною для широкого кола користувачів. Мікроконтролер ESP32 компанії Espressif Systems, один із найпоширеніших у сфері IoT, дозволяє підключати датчики безпосередньо через інтерфейс I²C та передавати зібрані дані через Wi-Fi-мережу без використання проміжних пристроїв. Датчик BME680 виробництва Bosch, що має компактний корпус розміром 3×3 мм, вимірює чотири параметри середовища одночасно, що робить його оптимальним вибором для вузлів моніторингу.

У запропонованій системі Raspberry Pi 5 виконує функції центрального сервера: приймає HTTP-запити від ESP32, зберігає виміряні значення у базі даних PostgreSQL та надає їх веб-клієнту. Такий підхід усуває залежність від зовнішніх хмарних сервісів, оскільки вся система функціонує у локальній мережі, а дані залишаються під контролем користувача.

Саме тому метою цієї роботи є розробка системи моніторингу мікроклімату приміщень, яка поєднує апаратний вимірювальний вузол на основі ESP32 + BME680, серверну частину на Spring Boot із зберіганням даних у PostgreSQL, а також веб-інтерфейс для відображення поточних і архівних показників.

Для досягнення поставленої мети визначено такі завдання:

- проаналізувати існуючі технічні рішення для моніторингу мікроклімату і визначити їхні обмеження;

- виконати порівняльний огляд бездротових технологій зв'язку та обґрунтувати вибір Wi-Fi і HTTP;
- розробити схему підключення датчика BME680 до ESP32 по протоколу I²C;
- написати прошивку мікроконтролера, що здійснює циклічне зчитування показань і відправлення їх на сервер за допомогою HTTP POST запитів;
- реалізувати серверну частину на Spring Boot з REST API для прийому, зберігання і видачі даних;
- розгорнути PostgreSQL і налаштувати схему бази даних для зберігання часових рядів вимірювань;
- розробити фронтенд-інтерфейс, який відображає поточні значення датчиків і графіки за обраний часовий проміжок;
- Провести тестування системи в реальних умовах з подальшою оцінкою точності та стабільності її роботи.

Історія Кафедри

Підготовка інженерів з засобів контролю і автоматичного керування технологічними процесами хімічної технології розпочата в Дніпропетровському хіміко-технологічному інституті в 1956 році на кафедрі загальної хімічної технології. Навчальний процес забезпечували доценти: Михайло Борисович Карцинель і Микола Микитович Мальцев, а також старший викладач Н.С. Янковська.

У 1959 році підготовка фахівців була передана окремо створеній кафедрі автоматизації виробничих процесів. Першим завідувачем кафедри був кандидат технічних наук, доцент Григорій Іванович Сипченко, що очолював її до 1966 року. Потім кафедрою завідували кандидати технічних наук, доценти: Михайло Борисович Карцинель (1966 – 1968 рр.), Микола Микитович Мальцев (1968 – 1976 рр.), Володимир Петрович Волосніков (1976 – 1982 рр.), Владислав Якович Тришкін (1982 – 2001 рр.), Олег Петрович Мисов (2001 – 2016 рр.), доктор технічних наук, професор Юрій Карлович Тараненко (2016 – 2018 рр.). З жовтня 2018 року її знову очолює кандидат технічних наук, доцент Мисов Олег Петрович.

З початку утворення на кафедрі основна увага була приділена двом основним напрямкам діяльності: методичному забезпеченню дисциплін, що викладаються, і науковим дослідженням із проблем автоматизації хіміко-технологічних процесів.

1964 рік

Кафедра здійснювала підготовку фахівців за спеціальністю “Автоматизоване керування технологічними процесами” і, з 1996 року – “Метрологія та вимірювальна

техніка". Поява другої спеціальності пов'язана зі стрімким збільшенням інформаційної складової частини автоматизованих систем керування технологічними процесами, виникненням високоточних і надійних методів і технічних засобів вимірів, впровадженням засобів обчислювальної техніки для вимірів і керування. За роки свого існування кафедра підготувала кілька тисяч фахівців в галузі автоматизації і вимірювальної техніки.

На 2023 рік навчальний процес з дисциплін кафедри здійснюють 2 професори, 7 доцентів, 2 старших викладача.

Кафедра обладнана сучасними засобами мікропроцесорної і вимірювальної техніки, має 2 комп'ютерних класи та 6 оснащених лабораторій.

Співробітники і випускники кафедри захистили 35 кандидатських і 6 докторських дисертацій, у тому числі: П.Г. Сорока – завідувач кафедри УДХТУ, В.З. Барсуков – завідувач кафедри Національного університету технологій і дизайну (Київ), В.І. Лісогор – завідувач кафедри Вінницького політехнічного інституту, Б.А. Блюсс – завідувач відділу ІГТМ (Дніпропетровськ), І.Г. Нестеров – генеральний директор ВО "Склопластик".

Перша група студентів для отримання в Українському державному хіміко-технологічному університеті спеціальності "**Спеціалізовані комп'ютерні системи**" була прийнята у 1996 році.

Особливості підготовки фахівців з цієї спеціальності викликали потребу в створенні окремої кафедри.

Тому в 1999 році на механічному факультеті УДХТУ на базі кафедри обчислювальної техніки та прикладної математики за ініціативою кандидата фізико-математичних наук, доцента Миколи Феодосійовича Огданського та професора Юрія Івановича Мережка була створена **кафедра інформаційних технологій та кібернетики**. Кафедру очолив кандидат фізико-математичних наук, доцент М.Ф. Огданський.

Кафедра отримала ліцензії на підготовку спеціалістів за спеціальністю "**Економічна кібернетика**", "**Спеціалізовані комп'ютерні системи**" та "**Інформаційні управляючі системи і технології**". В роки становлення вченим секретарем кафедри був кандидат технічних наук Євген Миколайович Логачов, а з 2002 року – старший викладач Світлана Вікторівна Бразинська.

У 2002-2003 роках усі спеціальності було акредитовано за освітньо-кваліфікаційним рівнем “спеціаліст”. У 2003 році кафедра здобула ліцензію на підготовку магістрів за спеціальністю “**Спеціалізовані комп’ютерні системи**“.

Активну участь в організації роботи кафедри приймали заступники завідувача кафедри, кандидат фізико-математичних наук Сергій Михайлович Заблуда (2002-2006) і кандидат хімічних наук Олександр Георгійович Капітонов (2006-2009).

До навчального процесу були залучені професори Е.В. Кочура, А.І. Федякін, Г.С. Пашковський, В.П. Пошивалов, В.М. Корчинський, О.І. Міхальов

З 2005 року естафету по розвитку спеціальності прийняв доктор технічних наук, професор Алпатов Анатолій Петрович, завідувач відділу системного аналізу та проблем управління Інституту технічної механіки Національної академії наук України і Національного космічного агентства України, керівник секції інформатики Придніпровського наукового центру Національної академії наук України і Міністерства освіти і науки України, засновник кафедри медичної кібернетики та обчислювальної техніки Дніпропетровської державної медичної академії.

У 2008 році на базі кафедри інформаційних технологій та кібернетики була створена **кафедра спеціалізованих комп’ютерних систем**, яка здійснює підготовку бакалаврів, спеціалістів та магістрів фахового напряму “**Комп’ютерна інженерія**” за спеціальністю “**Спеціалізовані комп’ютерні системи**“. В 2014 році кафедра СКС успішно пройшла чергову акредитацію бакалаврів, спеціалістів та магістрів та отримала відповідний сертифікат.

З 2010 року по 2014 кафедру СКС очолював доктор технічних наук, старший науковий співробітник Чумаков Лев Дмитрович. Він мав великий досвід роботи в організаціях Національної академії наук України та промисловості.

З листопада 2015 року кафедру СКС очолює доктор фізико-математичних наук, професор Анатолій Іванович Косолап. Він та його учні ведуть активну наукову роботу в галузі математичного моделювання та оптимізації складних систем. Розробляють програмне забезпечення, для проведення складних обчислень при проектуванні оптимальних систем. Отримані наукові результати знайшли відображення в 6-х монографіях, в яких пропонуються нові методи оптимізації складних систем, а також значна увага приділяється розробці ефективного програмного забезпечення. А.І. Косолап був учасником більше 40 міжнародних наукових

конференцій, як в Україні так і за її межами, де він доповідав нові наукові результати та обмінювався досвідом впровадження цих результатів в навчальний процес. Він є членом European Operational Research Societies. Значну увагу А.І. Косолап приділяє якості навчання та його відповідності кращим європейським та національним університетам. Кожен випускник кафедри повинен оволодіти сучасними методами моделювання складних комп'ютерних систем, оптимізувати їх структуру та функціонування за допомогою сучасного програмного забезпечення.

1. ЗАГАЛЬНИЙ РОЗДІЛ

1.1 Поняття та класифікація IoT-систем

Концепція «Інтернету речей» (Internet of Things, IoT) набула широкого вжитку на початку 2000-х років, хоча задум об’єднання фізичних об’єктів у мережу виник значно раніше. В інженерному розумінні IoT – це екосистема фізичних пристроїв, оснащених вбудованими сенсорами, мікроконтролерами та бездротовими інтерфейсами, що забезпечують автономний збір і обмін інформацією без участі оператора.

Галузі впровадження IoT охоплюють широкий спектр – від побутової автоматизації та носимих гаджетів до промислових SCADA-систем, точного землеробства і телемедицини. Незважаючи на різноманітність прикладних задач, переважна більшість IoT-рішень базується на уніфікованій багаторівневій архітектурі.

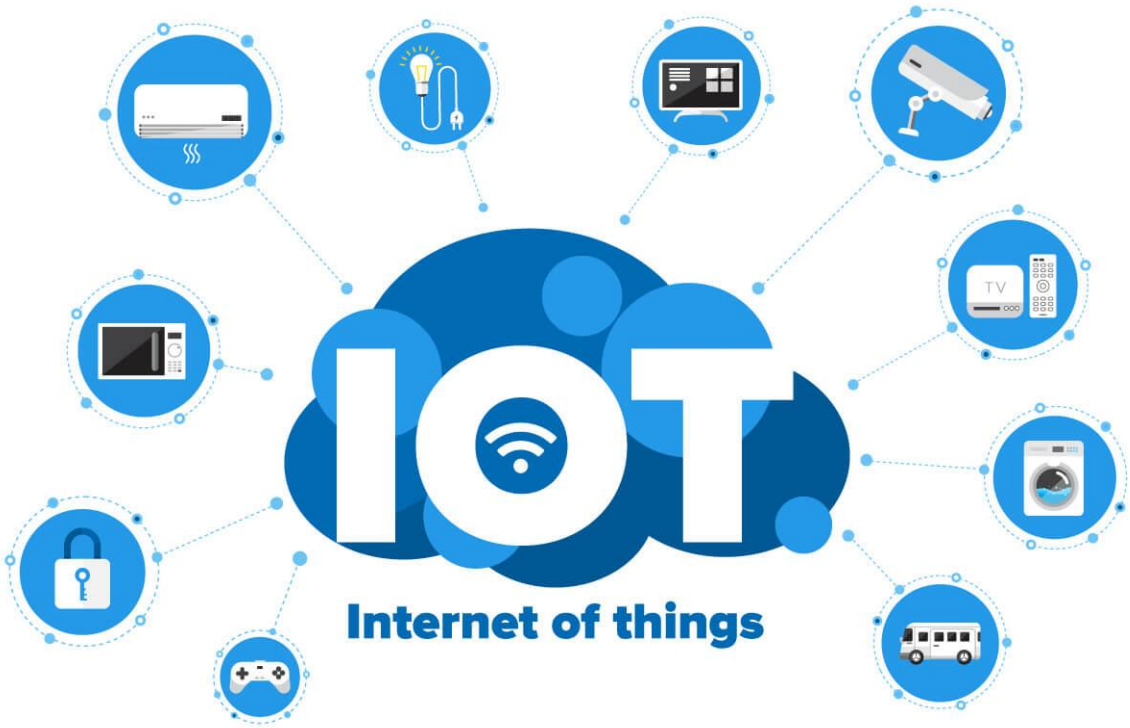


Рисунок 1.1 – Представлення IOT

Зазвичай в IOT використовують чотири рівневу архітектуру а саме:

Перший рівень сприйняття (Perception Layer) Це «органи чуття» системи. Сюди входять фізичні пристрої та датчики — смартфони, смарт-годинники, автомобілі, термометри, камери відеоспостереження, RFID-мітки. Вони збирають дані з навколишнього середовища та передають їх далі в систему.

Другий рівень Мережевий (Network Layer) Відповідає за передачу даних між пристроями. Включає протоколи бездротового зв'язку — Bluetooth, ZigBee, Wi-Fi, мобільні мережі 3G/4G/5G. Також на цьому рівні працює edge computing — попередня обробка даних безпосередньо поблизу джерела, що зменшує навантаження на мережу.

Третій рівень проміжний (Middleware Layer) Забезпечує обробку, зберігання та аналіз великих обсягів даних. Головний інструмент — хмарні обчислення (cloud computing). Хмара отримує дані від мільйонів пристроїв, аналізує їх і надає результати застосункам верхнього рівня.

Четвертий рівень застосунків (Application Layer) Кінцевий рівень, з яким взаємодіє користувач. Це мобільні додатки, комп'ютери, смарт-телевізори, планшети — все, що відображає результати роботи системи. Зв'язок з нижніми рівнями відбувається через інтернет.

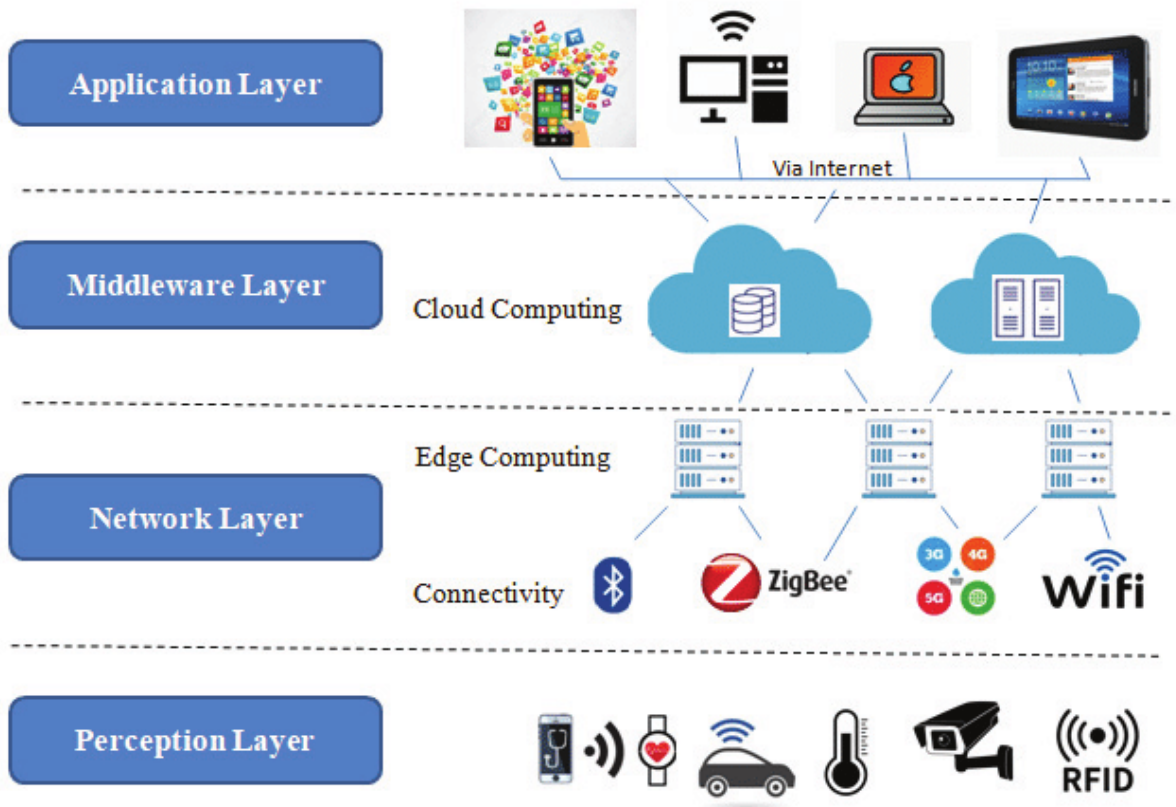


Рисунок 1.2 – Рівні IOT

За географічним охопленням IoT-мережі класифікують на персональні (WPAN, до 10 м – наприклад, Bluetooth між смартфоном і розумним годинником), локальні (WLAN, до 100 м – Wi-Fi у будинку чи офісі), міські (WMAN, до десятків кілометрів – технології LoRa, Sigfox) і глобальні (WWAN – NB-IoT, LTE-M через стільникових операторів). У даному проєкті реалізована WLAN-архітектура: ESP32 і сервер Raspberry Pi знаходяться в одній домашній Wi-Fi-мережі.

За сферою застосування виділяють такі класи IoT-систем: промисловий IoT– автоматизація виробничих процесів і передбачуване обслуговування обладнання; розумний будинок (Smart Home) – управління освітленням, кліматом, безпекою; розумне місто (Smart City) – моніторинг трафіку, витрат ресурсів, рівня забруднення; медичний IoT (IoMT) – дистанційний моніторинг пацієнтів і носимі пристрої. Розроблювана система відноситься до класу Smart Home / Smart Office.

Важливим аспектом IoT є забезпечення безпеки. Оскільки пристрої постійно передають дані по мережі, вони можуть бути потенційними векторами атак. Основні загрози: перехоплення даних (man-in-the-middle), несанкціонований доступ до API, атаки на вбудоване програмне забезпечення. У поточній реалізації система функціонує в ізольованій локальній мережі без виходу в інтернет, що суттєво знижує поверхню атаки. У перспективі передбачається додавання автентифікації API-токенами та перехід на HTTPS.

1.2 Огляд систем моніторингу мікроклімату

Мікроклімат приміщення характеризується сукупністю параметрів повітряного середовища, від яких залежить самопочуття і здоров'я людини. Основні з них – температура повітря, відносна вологість, тиск і рівень CO_2. Санітарні норми ДСН встановлюють оптимальні значення: температура 21–23 °C (влітку) і 22–24 °C (взимку), відносна вологість 40–60%, тиск – (101325) Па.

З практичного погляду найбільш критичними є три параметри. По-перше, температура – її відхилення навіть на кілька градусів від норми суттєво впливає на продуктивність і концентрацію. По-друге, вологість: занадто суха атмосфера подразнює слизові оболонки, а надмірна – сприяє розвитку цвілі і псуванню меблів. По-третє, атмосферний тиск – його різкі зміни особливо відчутні у людей зі серцево-судинними захворюваннями.

Для автоматизованого спостереження за цими параметрами використовуються різні класи пристроїв. Найпростіші – автономні термогігрометри з РК-дисплеєм – фіксують поточне значення, але не дають можливості переглядати архів. Наступний рівень – мережеві метеостанції з хмарним сервісом, наприклад Netatmo або Xiaomi Mi. Вони зручні у користуванні, але потребують постійного інтернет-з'єднання і прив'язані до екосистеми конкретного виробника. Нарешті, корпоративні системи такі як Grafana + InfluxDB + Telegraf забезпечують повний контроль над даними і гнучку аналітику, проте вимагають значних зусиль на налаштування.

Дана робота займає проміжну позицію між останніми двома класами: розроблювана система повністю автономна і не залежить від зовнішніх сервісів, але при цьому достатньо проста для розгортання.

Системи моніторингу мікроклімату пройшли довгий шлях від простих аналогових термометрів до складних інтегрованих платформ з хмарною аналітикою. Перші промислові системи автоматизованого моніторингу з'явилися у 1970-х роках у вигляді систем управління будівлями (Building Automation Systems, BAS), проте вони коштували десятки тисяч доларів і були доступні лише для великих промислових об'єктів. Поява дешевих мікроконтролерів і бездротових протоколів у 2010-х демократизувала цю галузь.

Для медичних і промислових застосувань до систем моніторингу мікроклімату висуваються суворі вимоги відповідно до стандартів ISO (чисті приміщення), GMP (фармацевтичне виробництво) та ДСН. Ці стандарти регламентують допустимі діапазони параметрів, метрологічні характеристики вимірювальних засобів, частоту калібрування і вимоги до резервного копіювання даних. Розроблювана система орієнтована на побутове та офісне застосування, де такі жорсткі вимоги не є обов'язковими.

Рисунок 1.3 – Складові виробничого мікроклімату

Ключовим трендом останніх років є інтеграція датчиків якості повітря. Окрім базових параметрів (температура, вологість, тиск), сучасні рішення відслідковують вміст CO2 (критичний для концентрації уваги), летких органічних сполук VOC (свіжа фарба, засоби для чищення), твердих частинок PM2.5 і PM10 (пил, алергени). Датчик BME680, обраний для даного проєкту, вимірює показник IAQ (Indoor Air Quality) – тобто інтегральну оцінку якості повітря на основі газового сенсора.

1.3 Аналіз існуючих комерційних рішень

Перед тим як проєктувати власну систему, потрібно доцільно оглянути те, що вже є на ринку. Це допоможе не «винаходити велосипед» і краще зрозуміти, яку нішу займатиме розроблювана система.

Netatmo Weather Station – французька метеостанція з двома блоками (вуличним і кімнатним). Передає дані через Wi-Fi у хмарний сервіс Netatmo, де є доступні графіки і API для інтеграцій. Головний недолік – без інтернету і серверів Netatmo система марна, а дані зберігаються у стороннього провайдера тому для автономного розгортання в локальній мережі ця система не підходить. Коштує приблизно 6000 грн.

Рисунок 1.4 – Netatmo Weather Station

Xiaomi Mi Temperature and Humidity Monitor 2 – дуже дешевий Bluetooth-датчик з екраном. Обмінюється даними зі шлюзом Mi Home по Bluetooth і дозволяє переглядати показання в додатку. Дальність дії і швидкість оновлення залишають бажати кращого, а архів зберігається тільки в китайській хмарі. Коштує 400 грн.

Рисунок 1.5 – Xiaomi Mi Temperature and Humidity Monitor 2

Grafana + InfluxDB – корпоративний стек для візуалізації часових рядів. Надзвичайно потужний, але явно надмірний для єдиного датчика в квартирі. Порогові витрати на розгортання і підтримку занадто великі.

Підсумовуючи: жодне з існуючих рішень не закриває потребу в простій, повністю автономній системі, яку можна розгорнути на звичайному одноплатному комп'ютері і підтримувати без спеціальних знань. Саме цю нішу й займає розроблювана система.

Для кількісного порівняння розглянутих рішень у таблиці 1.1 зведено ключові критерії вибору: автономність (незалежність від зовнішніх сервісів), вартість

розгортання, можливість доопрацювання, конфіденційність даних і підтримка власного сенсора.

Таблиця 1.1 – Порівняльний аналіз рішень для моніторингу мікроклімату

Рішення	Автоном-ність	Вартість	Доопра-цювання	Конфіден-ційність	Кастом. сенсор
Netatmo Weather Station	Низька	Висока	Неможливо	Низька	Неможливо
Xiaomi Mi Monitor 2	Середня	Низька	Неможливо	Низька	Неможливо
Grafana + InfluxDB	Висока	Висока	Можливо	Висока	-
Дана розробка (ESP32)	Повна	Низька	Можливо	Максимальна	Вбудовано

З таблиці видно, що тільки власна розробка забезпечує повну автономність, максимальну конфіденційність (дані ніколи не покидають локальну мережу), і при цьому є найбільш економічно доступним рішенням. Орієнтовна вартість апаратних компонентів становить близько 4 600 грн (ESP32 DevKit – 250 грн, BME680 модуль – 350 грн, Raspberry Pi 5 – 4 000 грн), тоді як комерційні системи з аналогічним функціоналом коштують від 400 до 6000 грн без можливості розширення.

1.4 Raspberry Pi 5 як обчислювальний вузол системи

Raspberry Pi 5 – одноплатний комп'ютер, представлений у жовтні 2023 року. Оснащений 4-ядерним процесором ARM Cortex-A76 з тактовою частотою 2,4 ГГц, що приблизно вдвічі-втричі швидше за попередника Raspberry Pi 4. Обсяг оперативної пам'яті – 4 або 8 ГБ DDR4, чого більш ніж достатньо для одночасного запуску Spring Boot, PostgreSQL і фонових процесів Linux.

З точки зору підключень: два порти USB 3.0 і два USB 2.0, Gigabit Ethernet, Wi-Fi, Bluetooth, PCIe 2.0 (новинка в серії, дозволяє підключити NVMe-накопичувач), два роз'єми Micro HDMI та 40-контактний GPIO. Важлива деталь – вперше в серії Pi з'явився вбудований роз'єм для батарейки RTC, що забезпечує збереження системного часу без мережі.

Для ролі HTTP-сервера і бази даних у локальній мережі ці характеристики є надлишковими з запасом. Проте саме такий запас гарантує, що система оброблятиме запити від кількох ESP32-вузлів одночасно без затримок, а також залишатиме ресурси для майбутнього розширення функціоналу.

Операційна система – Raspberry Pi OS (64-розрядна версія, Debian Bookworm). Підтримує JDK 21 та PostgreSQL 15+, що забезпечує повноцінну роботу всіх компонентів системи.

Для розгортання серверної частини системи обрано Raspberry Pi 5 завдяки оптимальному поєднанню обчислювальних ресурсів, низького енергоспоживання (5-15 Вт) і доступної вартості. На відміну від звичайного ПК, Raspberry Pi 5 споживає значно менше електроенергії при цілодобовій роботі, займає мінімум місця і практично безшумний при використанні з охолоджувальним корпусом.

Таблиця 1.2 – Технічні характеристики Raspberry Pi 5

Параметр	Значення
Процесор	Broadcom BCM2712, 4-ядерний ARM Cortex-A76,
Оперативна пам'ять	4гб DDR4
Мережа	Gigabit Ethernet, Wi-Fi 802.11ac, Bluetooth 5.0
USB	2x USB 3.0 (5 Гбіт/с), 2x USB 2.0
GPIO	40-контактний роз'єм (PWM, I2C, SPI, UART)
Накопичувач	microSD + PCIe 2.0 FPC для NVMe SSD
Живлення	USB-C 5V/5A (25 Вт максимум)
Розміри	85 x 56 x 17 мм
Споживання в роботі	5–15 Вт залежно від навантаження
Операційна система	Raspberry Pi OS 64-bit, JDK 21, PostgreSQL 15+

Продуктивність ARM Cortex-A76 на частоті 2.4 ГГц є достатньою для одночасного виконання Spring Boot (JVM), PostgreSQL і обробки REST API запитів без помітних затримок. За результатами тестування, відповідь на агрегаційний запит (average за 24 години) становить менше 50 мс, що значно краще встановленої вимоги 300 мс.

Рисунок 1.6 – Raspberry Pi 5 у захисному корпусі із системою охолодження

1.5 Мікроконтролер ESP32: характеристики та можливості

ESP32 – серія мікроконтролерів компанії Espressif Systems, що вийшла у 2016 році і швидко стала стандартом для IoT-розробки. Основна версія містить два ядра Xtensa LX6 з частотою до 240 МГц, 520 КБ SRAM і вбудований Wi-Fi разом із Bluetooth.

Для підключення датчиків ESP32 надає повний набір інтерфейсів: два апаратних I²C, три SPI, три UART, два 12-бітних АЦП, два ЦАП, підтримку ємнісних сенсорних

входів та PWM. Це означає, що до одного мікроконтролера можна підключити одночасно кілька різних датчиків без додаткових мікросхем-розширювачів.

Wi-Fi реалізований апаратно і підтримує режими Station (підключення до існуючої мережі), Access Point (власна точка доступу) і їхнє поєднання. У даному проєкті ESP32 працює у режимі Station: підключається до домашньої Wi-Fi мережі і надсилає HTTP POST запити на IP-адресу Raspberry Pi.

Для програмування використовується ESP-IDF – офіційний фреймворк від Espressif на базі FreeRTOS. Він дає доступ до всіх апаратних можливостей і надає готові компоненти для роботи з I²C, HTTP-клієнтом, Wi-Fi і іншим. Збірка проєкту виконується через CMake і Ninja. Альтернативою є Arduino-фреймворк, але ESP-IDF обраний як більш стабільний і гнучкий.

Вартість ESP32-модуля у форм-факторі DevKit, робить рішення економічно привабливим для розгортання одразу в кількох кімнатах.

Таблиця 1.3 – Технічні характеристики мікроконтролера ESP32

Параметр	**Значення**
Процесор	Dual-core Xtensa LX6, до 240 МГц
Пам'ять SRAM	520 КБ (на чіпі) + зовнішня Flash до 16 МБ
Wi-Fi	802.11 b/g/n (2.4 ГГц), Station / AP / Monitor
Bluetooth	BT 4.2 Classic + BLE
GPIO	34 контакти (PWM, АЦП, ЦАП, I2C, SPI, UART)
I²C	2 апаратних контролери (будь-які GPIO)
АЦП	18 каналів 12-біт
Живлення GPIO	3.3 В логіка, VIN 2.3–3.6 В
Споживання (Wi-Fi активний)	~170 мА при 3.3 В
Споживання (Deep Sleep)	~10 мкА
Температурний діапазон	-40 ... +85 °C
Ціна модуля DevKit	3–5 USD (~250 грн)

Ключовою перевагою ESP32 є два незалежних ядра. Перше ядро (Pro CPU) виконує Wi-Fi та Bluetooth стек під управлінням ESP-IDF. Друге ядро (App CPU) вільне для користувацьких завдань – у нашому випадку це циклічне зчитування BME680 і формування HTTP-запитів. FreeRTOS, що є основою ESP-IDF, дозволяє запускати кілька паралельних задач з пріоритетами і семафорами для захисту спільних ресурсів.

Режими енергозбереження ESP32 дозволяють значно зменшити споживання при автономному живленні від акумулятора. Deep Sleep зупиняє основні ядра і всю периферію, залишаючи активним лише таймер RTC і окремі GPIO. У даному проєкті ESP32 живиться від мережі через USB і постійно активний, проте підтримка Deep Sleep відкриває можливість переходу на батарейне живлення у майбутньому.

Рисунок 1.7 – Модуль ESP32 DevKit з вбудованим Wi-Fi/BT SoC (ISM 2.4G 802.11b/g/n)

1.6 Датчик BME680: принцип роботи та технічні характеристики

BME680 – інтегрований датчик навколишнього середовища компанії Bosch Sensortec, що в одному корпусі розміром 3×3×0,93 мм об'єднує чотири вимірювальні системи: температури, відносної вологості, атмосферного тиску і резистивного газового датчика для оцінки якості повітря (IAQ – Indoor Air Quality).

Усередині датчика три незалежних вимірювальних елементи. Температуру і вологість вимірює мікроскопічний конденсатор, ємність якого змінюється залежно від вологості і нагріву – за цією зміною і визначаються значення. Тиск вимірюється крихітною мембраною, яка прогинається під вагою повітря. Газовий датчик – це маленький нагрівач з резистором: при різних запахах і газах опір змінюється, і за цією зміною можна судити про якість повітря. У даному проєкті задіяні всі чотири канали датчика: температура, вологість, тиск і газовий опір (IAQ) – всі вони відображаються на веб-інтерфейсі у вигляді окремих графіків.

Датчик працює у режимі Forced Mode – перед кожним вимірюванням вбудований нагрівач прогрівається до робочої температури, після чого знімаються показники з усіх каналів одночасно.

Таблиця 1.4 – Основні технічні характеристики датчика BME680

Параметр	Значення / діапазон	Примітка
Живлення (VDD)	1.71 – 3.6 В	підтримує 3.3
Вимірювані величини	T, RH, P, IAQ	температура, вологість, тиск, якість повітря
Інтерфейс	I²C / SPI	для даного проєкту обрано I²C
Адреса I²C	0x76 або 0x77	залежить від рівня SDO
Точність температури	±1.0 °C	діапазон –40…+85 °C
Точність вологості	±3% rH	діапазон 0…100% rH
Точність тиску	±1.7 гПа	діапазон 300…1100 гПа
Час відгуку (T/RH)	1 с / 8 с	при оновленні 1 Гц
Споживання у роботі	~3.7 мА	в режимі вимірювань

Для роботи з датчиком з боку ESP32 використовується офіційна бібліотека Bosch BME68x Sensor API, адаптована під ESP-IDF. Вона забезпечує ініціалізацію датчика, вибір режиму вимірювань і зчитування скомпенсованих значень у фізичних одиницях.

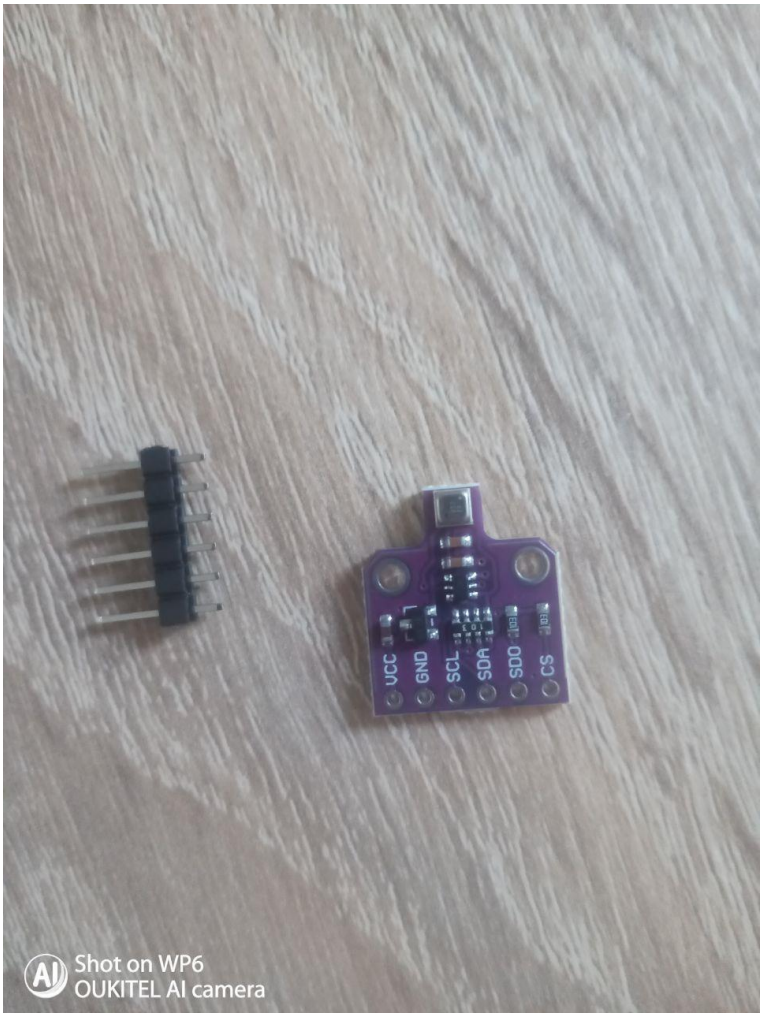


Рисунок 1.8 – Модуль датчика BME680

2. ОСНОВНИЙ РОЗДІЛ

2.1 Порівняльний аналіз бездротових технологій для IoT

2.1.1 Технології LPWAN

Коли IoT-пристрої розміщуються на значних відстанях одне від одного або всередині будівель зі складною планіровкою, традиційний Wi-Fi з його радіусом дії 30–50 м і помітним споживанням енергії може виявитися невідповідним вибором.

Саме тому на початку 2010-х з'явився клас технологій LPWAN (Low Power Wide Area Network) – мережі далекого радіусу дії з низьким споживанням.

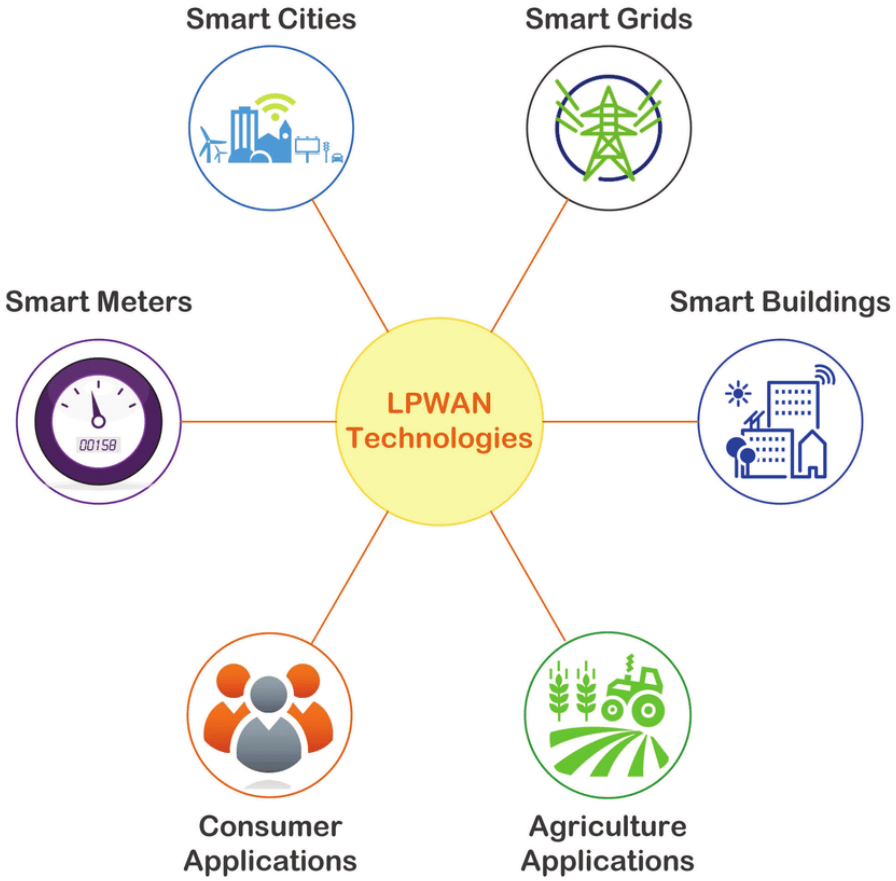


Рисунок 2.1 – Представлення LPWAN

LoRa (Long Range) – технологія для передачі даних на великі відстані з мінімальним споживанням енергії. Може «добити» до 15 км у відкритому полі. Звучить чудово, але для нашого завдання є два мінуси: передача даних повільна (грубо кажучи, можна надіслати лише кілька коротких повідомлень на секунду), і потрібен окремий шлюз – додатковий пристрій між датчиком і сервером.

SigFox – ще одна технологія для дуже далеких відстаней, але з жорстким обмеженням: один пристрій може надіслати не більше 140 повідомлень на добу. Для датчика що передає дані кожну хвилину це катастрофічно мало – 140 повідомлень вичерпається вже за перші дві години.

NB-IoT – це IoT через мобільну мережу стільникових операторів. Покриття є скрізь де є мобільний зв'язок. Але є суттєвий мінус: потрібна SIM-картка і щомісячна абонентська плата оператору. Для домашньої системи це надлишково.

2.1.2 Технології ближнього радіусу (Wi-Fi, Zigbee, BLE)

Для системи в межах одного будинку або офісу технології LPWAN надмірні. Тут конкурують декілька протоколів ближнього і середнього радіусу.

Wi-Fi на сьогодні присутній практично в кожному житловому і офісному приміщенні, а отже, не вимагає додаткової інфраструктури. Стандарт який забезпечує пропускну здатність у сотні мегабіт на секунду – для передачі кількох десятків байт вимірювань це колосальний запас. Важливо, що TCP/IP-стек Wi-Fi дозволяє безпосередньо відправляти HTTP-запити, не вимагаючи ніяких проміжних шлюзів або додаткових протоколів.

Zigbee – технологія яку використовують у розумних лампочках Philips Hue і меблях IKEA. Економна, дальність нормальна. Мінус для нашого випадку: щоб Zigbee-пристрій міг «говорити» з сервером, потрібен окремий координатор – ще один пристрій, який треба купити і налаштувати.

Bluetooth Low Energy (BLE) – технологія з дуже низьким споживанням, радіусом до 40 м і швидкістю до 2 Мбіт/с у версії Bluetooth 5.1/5.2 підтримує mesh-мережі, що відкрило нові можливості для IoT-застосувань. Для зв'язку з сервером також потрібен шлюз (смартфон, Raspberry Pi з Bluetooth-адаптером).

Таблиця 2.1 – Порівняльний аналіз бездротових технологій

Критерій	Wi-Fi (ESP32)	Zigbee	LoRa	BLE 5.x
Радіус дії (в приміщенні)	30–50 м	10–75 м	до 15 км	10–40 м
Споживання енергії	помірне	низьке	дуже низьке	дуже низьке
Швидкість передачі	до 150 Мбіт/с	до 250 кбіт/с	0.3–50 кбіт/с	до 2 Мбіт/с
Вартість модуля	низька	середня	середня	низька
Потреба у шлюзі	ні	так (координатор)	так (шлюз)	так (для mesh)
TCP/IP підтримка	так	ні	ні	обмежена
Придатність для проєкту	висока	середня	низька	середня

2.1.3 Обґрунтування вибору Wi-Fi та HTTP

З таблиці 2.1 видно, що для умов даного проєкту Wi-Fi є найбільш привабливим варіантом за сукупністю факторів. Ключові аргументи:

- ESP32 має вбудований Wi-Fi-контролер – не потрібен додатковий модуль і немає ускладнення схеми;
- існуюча домашня мережа повністю покриває всі потрібні точки розміщення датчиків;
- TCP/IP-стек дозволяє використовувати HTTP безпосередньо, без проміжних брокерів чи шлюзів;
- налагодження HTTP-запитів значно простіше, ніж бінарних протоколів – можна перевірити будь-яким браузером або curl;
- бібліотека HTTP-клієнта у складі ESP-IDF добре задокументована і стабільна.

Відносно більше споживання Wi-Fi порівняно з Zigbee або BLE у даному застосуванні не є проблемою: ESP32-модуль живиться від мережі (USB-адаптер), тому автономність від батареї не потрібна.

2.2HTTP як протокол передачі даних між ESP32 і сервером

HTTP (HyperText Transfer Protocol) – протокол прикладного рівня, що визначає правила обміну повідомленнями між клієнтом і сервером у мережах TCP/IP. У розробленій системі мікроконтролер ESP32 виступає HTTP-клієнтом, що ініціює з’єднання, а серверна частина на Raspberry Pi – HTTP-сервером, що приймає запити та повертає відповіді.

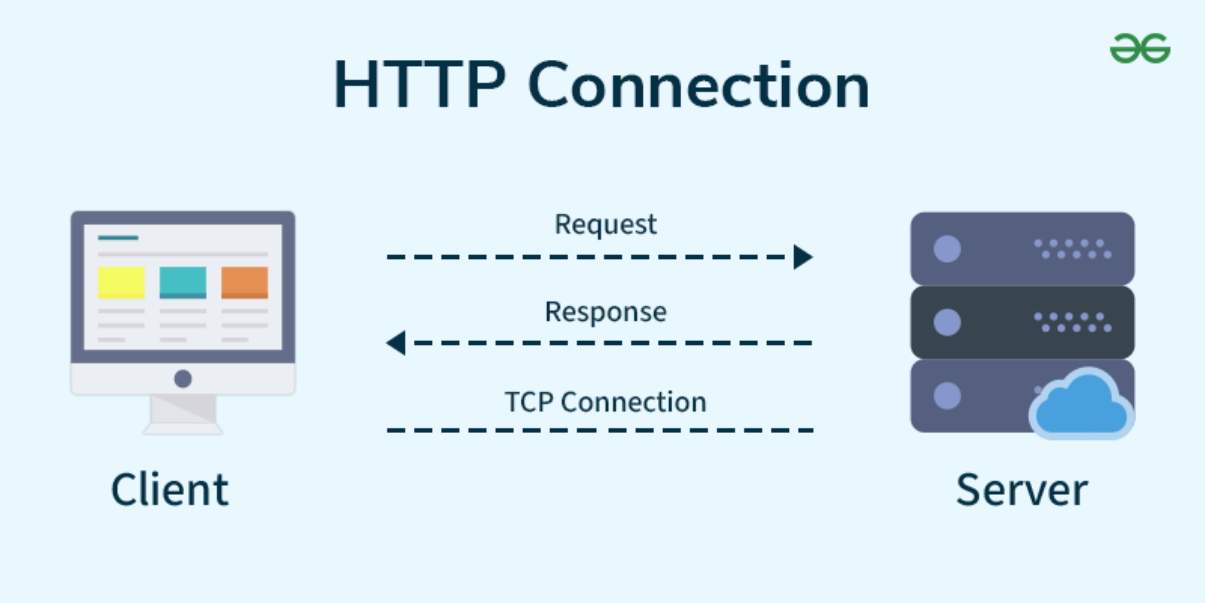


Рисунок 2.2 – Схема роботи http

Взаємодія між ESP32 і Raspberry Pi у системі реалізована таким чином. Мікроконтролер після кожного циклу вимірювань формує JSON-рядок із поточними значеннями датчика і відправляє HTTP POST запит на ендпоїнт Spring Boot сервера. Наприклад: сервер, отримавши запит, десеріалізує JSON, записує дані в PostgreSQL і повертає HTTP 200 OK. Такий підхід надзвичайно простий у налагодженні і не вимагає спеціального клієнтського програмного забезпечення – будь-який HTTP-клієнт може протестувати сервер незалежно від ESP32.

Окрім ендпоїнту запису, сервер надає GET-ендпоїнти для фронтенду. Наприклад:

- /api/measurements/latest – поточні значення датчиків (JSON з останніми записами);
- /api/measurements?from=...&to=... – вибірка за часовим діапазоном;
- /api/measurements/average?period=hour – середні значення за заданий період.

Саме ці ендпоїнти і споживає фронтенд-застосунок для побудови графіків і відображення поточного стану. Детальний опис усіх ендпоїнтів наведено в розрахунковому розділі.

Слід зазначити, що у виробничій системі з вимогами до безпеки рекомендується перейти на HTTPS і додати автентифікацію. У рамках навчального прототипу, що працює в ізольованій локальній мережі, незахищений HTTP є прийнятним рішенням.

У реалізованій системі кожен HTTP-запит від ESP32 включає три елементи: стартовий рядок з методом POST і шляхом /api/bme680/microclimate, заголовок Content-Type: application/json що повідомляє серверу формат тіла, і JSON-тіло з чотирма полями вимірювань. Spring Boot при успішному збереженні повертає код 201 Created разом із серіалізованим об'єктом запису — включно з автоматично присвоєним ідентифікатором і міткою createdAt. ESP32 перевіряє лише клас відповіді: будь-який 2xx-код трактується як підтвердження успішної передачі, після чого з'єднання закривається і ресурси звільняються.

Метод HTTP POST використовується тоді, коли клієнт хоче створити новий ресурс на сервері – саме це і робить ESP32, надсилаючи нові вимірювання. Метод GET, навпаки, лише отримує дані без жодних змін. REST-принцип ідемпотентності вимагає: GET-запити не повинні змінювати стан системи і при повторному виклику дають той самий результат, тоді як кожен POST може створювати новий запис.

Коди HTTP-статусу несуть важливу інформацію про результат операції. У системі використовуються: 200 OK – запит виконаний успішно (GET-запити); 201 Created – новий ресурс створено (відповідь на POST від ESP32); 400 Bad Request – неправильний формат JSON або відсутні обов'язкові поля; 404 Not Found – запис із вказаним ID не знайдено; 500 Internal Server Error – несподівана помилка на стороні сервера (наприклад, недоступна база даних).

JSON (JavaScript Object Notation) – текстовий формат обміну даними на основі двох структур: об'єкт і масив. У проєкті ESP32 формує такий JSON-рядок:

{"temperature": 20.45,

"humidity": 58.3,

"pressure_hpa": 1013.2,

"gas_resistance_ohm": 15420}.

Spring Boot автоматично десеріалізує цей JSON у Map<String, Object> і зберігає як значення типу JSONB у PostgreSQL.

2.3 Серверна частина: Spring Boot і Java 21

Spring Boot – фреймворк для розробки серверних застосунків на платформі Java, що реалізує принцип «convention over configuration»: більшість налаштувань

застосовуються автоматично на основі залежностей у classpath, що усуває необхідність ручного XML-конфігурування. Вбудований контейнер сервлетів Tomcat дозволяє упаковувати застосунок у єдиний виконуваний JAR-файл, придатний для безпосереднього запуску на цільовому сервері.

Вибір Java 21 (Long-Term Support – версія з тривалою офіційною підтримкою до 2031 року) обумовлений кількома факторами. По-перше, це гарантія стабільних оновлень безпеки на роки вперед. По-друге, Java 21 принесла Virtual Threads (проєкт Loom) – «легкі» потоки виконання, якими керує JVM, а не операційна система. Простіше кажучи, сервер може одночасно обробляти сотні HTTP-запитів, не витрачаючи оперативну пам'ять на створення повноцінного системного потоку для кожного. По-третє, ARM64-збірки JDK 21 (процесорна архітектура Raspberry Pi 5) офіційно підтримуються і добре протестовані.

Серверний код організований за чотирирівневою архітектурою, де кожен рівень має суворо визначену відповідальність і не знає деталей реалізації сусіднього рівня.

Controller (рівень контролерів) отримує вхідні HTTP-запити, десеріалізує параметри і делегує виклик до сервісного рівня. Контролер не містить бізнес-логіки – він лише маршрутизує запити.

Service (сервісний рівень) реалізує бізнес-логіку: перевірку коректності даних, агрегацію і перетворення результатів перед поверненням клієнту.

Repository (рівень репозиторіїв) інкапсулює всю взаємодію з базою даних. Решта коду не залежить від конкретної СУБД і не знає деталей SQL-запитів – це дозволяє замінити PostgreSQL на іншу базу без змін у сервісному та контролерному рівнях.

Entity і DTO (Data Transfer Objects) – прості Java-класи, що описують структуру рядків бази даних та формат JSON-відповідей.

Apache Maven автоматично завантажує всі необхідні бібліотеки і збирає готовий виконуваний файл, що містить як код застосунку, так і всі його залежності. Такий підхід спрощує розгортання: на сервері достатньо запустити один файл без попередньої установки додаткових компонентів.

Таблиця 2.2 – Компоненти серверної частини та використані бібліотеки

Компонент	Бібліотека / анотація	Призначення
Головний клас застосунку	@SpringBootApplication (Spring Boot Autoconfigure)	Запускає застосунок; автоматично налаштовує HTTP-сервер Tomcat, підключення до БД і сканування компонентів — без ручного XML-конфігурування.
Клас-сутність Bme680Reading	@Entity, @Table (Hibernate / Jakarta Persistence)	Описує структуру таблиці bme680_readings як Java-клас. Hibernate сам перетворює рядки таблиці на об'єкти і навпаки.
Репозиторій Bme680ReadingRepository	JpaRepository, @Query (Spring Data JPA)	Надає готові методи збереження та читання (save, findAll, findById). Власні SQL-запити додаються через анотацію @Query.
Контролер Bme680ReadingController	@RestController, @CrossOrigin (Spring Web MVC)	Приймає HTTP-запити від браузера і повертає відповіді у форматі JSON. @CrossOrigin дозволяє браузеру звертатися до API з іншого порту (CORS).
Файл конфігурації application.yml	Spring Boot	Налаштування: порт 8081, рядок підключення до PostgreSQL, режим оновлення схеми БД. Змінюється без перекомпіляції.
Система збірки Apache Maven	pom.xml	Завантажує бібліотеки автоматично. Залежності: spring-boot-starter-web (Tomcat + REST), spring-boot-starter-data-jpa (Hibernate), postgresql (JDBC-драйвер).

2.4 База даних PostgreSQL

PostgreSQL – об'єктно-реляційна система управління базами даних з відкритим кодом, розвиток якої ведеться з 1986 року. Відрізняється суворим дотриманням стандартів SQL і потужною підтримкою транзакцій за принципом ACID, що гарантує цілісність даних навіть при збоях. Підтримує широкий набір типів даних, включаючи JSON, масиви і геометричні типи.

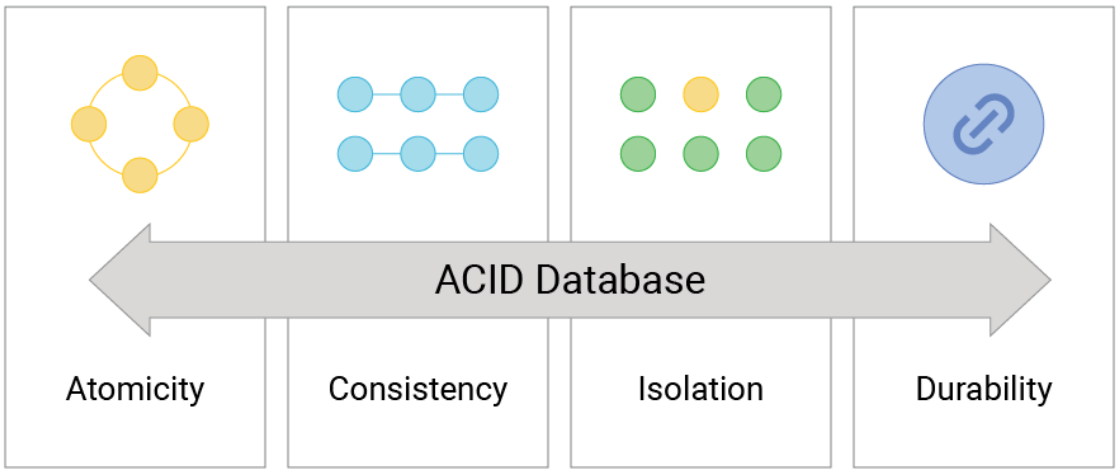


Рисунок 2.3 – Абревіатура ACID

Абревіатура ACID описує чотири фундаментальні гарантії надійності транзакційних систем. Атомарність (Atomicity) означає, що транзакція є неподільною одиницею: або всі її операції успішно завершуються, або жодна зміна не зберігається – проміжних станів не існує.

Узгодженість (Consistency) забезпечує дотримання всіх визначених обмежень схеми під час кожної транзакції: якщо операція намагається порушити обмеження цілісності – наприклад, вставити запис з дублюючим ідентифікатором – СУБД відхиляє її цілком. Для системи моніторингу це виключає появу некоректних або неповних записів вимірювань.

Ізоляція (Isolation) вирішує задачу коректного паралельного доступу до даних. У системі моніторингу ESP32 постійно записує нові вимірювання, тоді як веб-клієнт одночасно виконує читання архіву – ці операції не повинні впливати одна на одну. PostgreSQL реалізує ізоляцію через механізм версійності MVCC (Multi-Version

Concurrency Control), при якому кожна транзакція отримує власний знімок стану бази й не бачить незафіксованих змін від конкурентних транзакцій.

Довговічність (Durability) гарантує збереження підтверджених даних навіть при аварійному завершенні роботи. Технічно це досягається через попереднє журналювання WAL (Write-Ahead Log): перш ніж транзакція вважається завершеною, усі зміни фіксуються у журналі на постійному носії. При наступному запуску після збою PostgreSQL автоматично відновлює стан бази з WAL-журналу без втрати підтверджених записів.

Важливо також розуміти різницю між реляційними СУБД (такими як PostgreSQL) і нереляційними (NoSQL). Реляційні бази зберігають дані у пов'язаних таблицях і гарантують ACID-властивості – це підходить для даних з чіткою структурою, де важлива цілісність. NoSQL-бази (MongoDB, Redis, Cassandra) жертвують частиною ACID-гарантій заради масштабованості і гнучкості схеми. Для системи моніторингу з часовими рядами, де дані надходять регулярно і треба гарантувати їх збереження, PostgreSQL є оптимальним вибором.

Чому PostgreSQL, а не MySQL? Обидві бази непогані, але PostgreSQL краще справляється з даними що записуються постійно з часовою міткою – а саме так і виглядають вимірювання датчика. При великому архіві (скажімо, рік щохвилинних даних – це понад 500 тисяч записів) PostgreSQL зберігає швидкість запитів там, де MySQL починає гальмувати.

PostgreSQL встановлюється безпосередньо на Raspberry Pi OS як системний сервіс. Після встановлення пакета через менеджер пакетів служба реєструється в systemd і автоматично запускається при кожному завантаженні системи. Дані зберігаються у стандартній директорії файлової системи, а резервне копіювання виконується штатними засобами PostgreSQL.

Дані, що збираються системою, відносяться до класу часових рядів (time series). Часовий ряд – це послідовність вимірювань, де кожен запис прив'язаний до конкретного моменту часу. Особливість часових рядів полягає в тому, що дані накопичуються дуже швидко: при кроці 5 секунд за одну добу утворюється 8640 записів, за рік – понад 3,1 мільйона. Це вимагає спеціальних підходів до зберігання і запитів.

Для роботи з часовими рядами критично важливою є агрегація – зведення великої кількості детальних значень до узагальнених показників за певний період. Наприклад, замість 360 окремих вимірювань за годину зручніше бачити одне середнє значення. PostgreSQL надає для цього функцію date_bin(), яка «розбиває» вісь часу на рівні інтервали будь-якої тривалості і дозволяє обчислити середнє, мінімум або максимум для кожного інтервалу в одному SQL-запиті.

Також date_bin(interval, timestamp, origin) дозволяє округляти мітки часу до границі заданого інтервалу. Наприклад 14:03:25 округлить до 14:00:00 і 14:08:47 до 14:05:00. Це дозволяє групувати вимірювання по довільних часових бінах і агрегувати (AVG, MIN, MAX) показання в межах кожного біна. На відміну від date_trunc(), date_bin підтримує довільні інтервали не прив'язані до стандартних одиниць часу.

Схема бази даних включає таблицю вимірювань з такими полями: ідентифікатор (BIGSERIAL – це автоінкрементне ціле число типу bigint, яке PostgreSQL присвоює автоматично), часова мітка (TIMESTAMP WITH TIME ZONE – зберігає момент вимірювання разом із часовим поясом, щоб уникнути плутанини при зміні налаштувань сервера), значення температури, вологості та тиску (NUMERIC – числа довільної точності, без заокруглень), а також ідентифікатор пристрою (VARCHAR – рядок для імені ESP32-вузла). Поле з ідентифікатором пристрою зроблено навмисно, щоб у майбутньому легко додати другий і третій датчик у різних кімнатах без змін схеми.

Ключовою особливістю схеми є використання типу JSONB замість традиційних реляційних стовпців. JSONB (JSON Binary) – бінарне представлення JSON у PostgreSQL, яке на відміну від типу JSON зберігає дані в розібраному вигляді: це прискорює запити і дозволяє індексування вмісту. Головна перевага: при додаванні нових параметрів датчика не потрібно змінювати схему таблиці – новий параметр просто з'являється у JSON-об'єкті.

Таблиця 2.3 – Порівняння JSON і JSONB у PostgreSQL

Характеристика	JSON	JSONB
Зберігання	Текстовий рядок	Бінарний розібраний формат
Швидкість запису	Швидше	Повільніше (розбір при записі)
Швидкість читання/фільтрації	Повільніше	Значно швидше
Індекси GIN / GiST	Не підтримуються	Повна підтримка
Оператори @>, @@	Обмежено	Повна підтримка
Рекомендується для	Зберігання без запитів	Фільтрації і аналізу

2.5 Фронтенд: JavaScript та React

Для реалізації фронтенду обрано React – бібліотеку для побудови інтерактивних інтерфейсів від Meta. React використовує компонентну модель: інтерфейс розбивається на незалежні компоненти, кожен з яких відповідає за свій фрагмент UI і окремо управляє станом. Це полегшує розробку і повторне використання елементів.

Одна з головних переваг React – він не перемальовує всю сторінку щоразу коли змінилося одне число. Він відстежує що саме змінилося і оновлює тільки цей шматочок.

Щоб отримувати дані з сервера, фронтенд використовує вбудований браузерний API – Fetch API. Це стандартний засіб виконання HTTP-запитів, що не потребує підключення зовнішніх бібліотек. За потреби можна передати параметри (діапазон дат, інтервал агрегації, тип значення) і отримати JSON-відповідь від Spring Boot сервера.

Для побудови графіків використовується бібліотека Apache ECharts (підключена через пакет echarts-for-react). ECharts – потужний рушій для інтерактивної візуалізації даних від Apache Software Foundation. Він дозволяє легко будувати стовпчасті діаграми з підказками при наведенні, масштабуванням (slider та pinch) і адаптивним розміром. У реалізованому інтерфейсі кожен параметр датчика (температура, вологість, тиск, газовий опір) відображається на окремій діаграмі типу "bar".

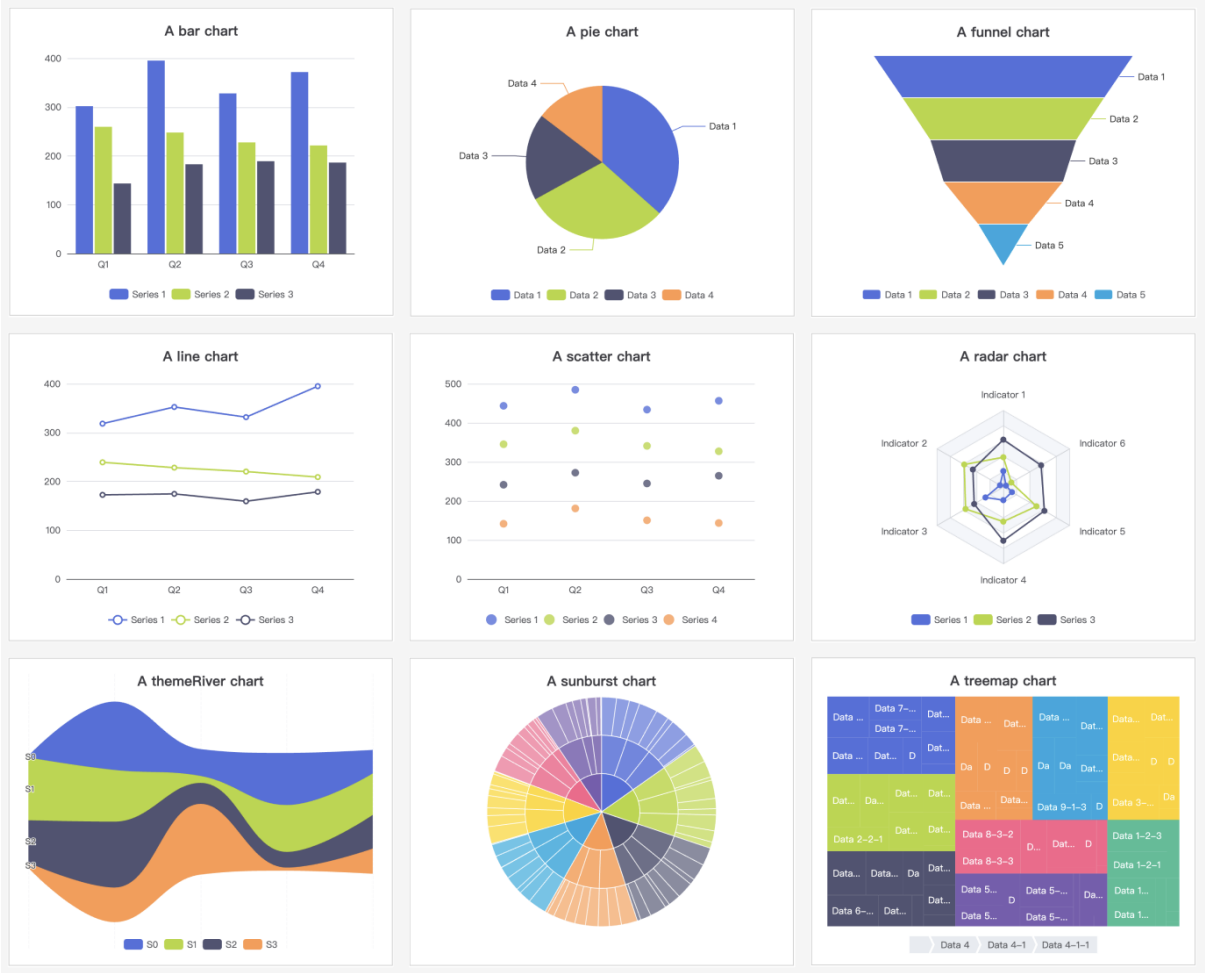


Рисунок 2.3 – Можливості ECharts

Застосунок реалізований за принципом SPA: після початкового завантаження сторінка більше не перезавантажується – React динамічно оновлює лише ті частини інтерфейсу, дані яких змінились. Інтерфейс складається з двох основних частин: панелі керування з полями налаштування запиту та набору чотирьох інтерактивних графіків – по одному для кожного параметра датчика.

Стан інтерфейсу – обраний діапазон дат, крок агрегації, режим відображення і завантажені дані – зберігається в пам'яті React-компонента і автоматично оновлює відповідні елементи сторінки при зміні. При натисканні кнопки "Показати параметри"

формується HTTP GET-запит до серверного API з усіма обраними параметрами; отримана JSON-відповідь розподіляється між чотирма графіками.

Vite – сучасний інструмент збірки, що використовує нативні ES-модулі браузера під час розробки.

2.6 Середовище розробки та інструментарій

Node.js v20 LTS і менеджер пакетів npm використовувались як середовище виконання для інструментів розробки React-застосунку. Ключова роль Node.js у даному проєкті — запуск Vite dev-сервера під час розробки і виконання команди vite build для фінальної збірки. Після збірки директорія dist із статичними файлами копіюється до ресурсів Spring Boot, після чого сервер роздає і фронтенд, і REST API на одному порту 8081 — без потреби в окремому Nginx або Apache.

Raspberry Pi OS 64-bit (Debian Bookworm) слугувалє серверним середовищем виконання. Критичною вимогою була саме 64-розрядна версія: 32-розрядна збірка не підтримує JDK 21, необхідний для Spring Boot. Встановлення JDK 21 і PostgreSQL 15 виконувалось через стандартний apt без додаткових налаштувань. Служби реєструвались у systemd з директивами After=postgresql.service і Restart=on-failure — це забезпечило автоматичний запуск у правильному порядку після увімкнення пристрою.

IntelliJ IDEA Community Edition використовувалась як основне середовище розробки серверної частини. Серед практично корисних функцій — вбудований HTTP-клієнт (.http-файли), що дозволяв тестувати ендпоїнти REST API безпосередньо з IDE без Postman: наприклад, перевіряти коректність десеріалізації JSON від ESP32 і формат відповіді на GET-запит агрегації. Для фронтенду застосовувався Visual Studio Code з розширеннями ESLint і Prettier, які автоматично виправляли форматування при збереженні файлу.

ESP-IDF Installation Manager — графічна утиліта для керування встановленими версіями ESP-IDF SDK і відповідними інструментальними ланцюжками компілятора. Вона дозволяє встановлювати кілька версій SDK паралельно, перемикатись між ними та оновлювати компоненти без ручного редагування змінних середовища. У даному проєкті використовувалась версія ESP-IDF v6.0.1 з інструментальним ланцюжком

Xtensa LX6 — саме ця конфігурація забезпечує підтримку всіх задіяних компонентів: I²C, esp_http_client і cJSON.

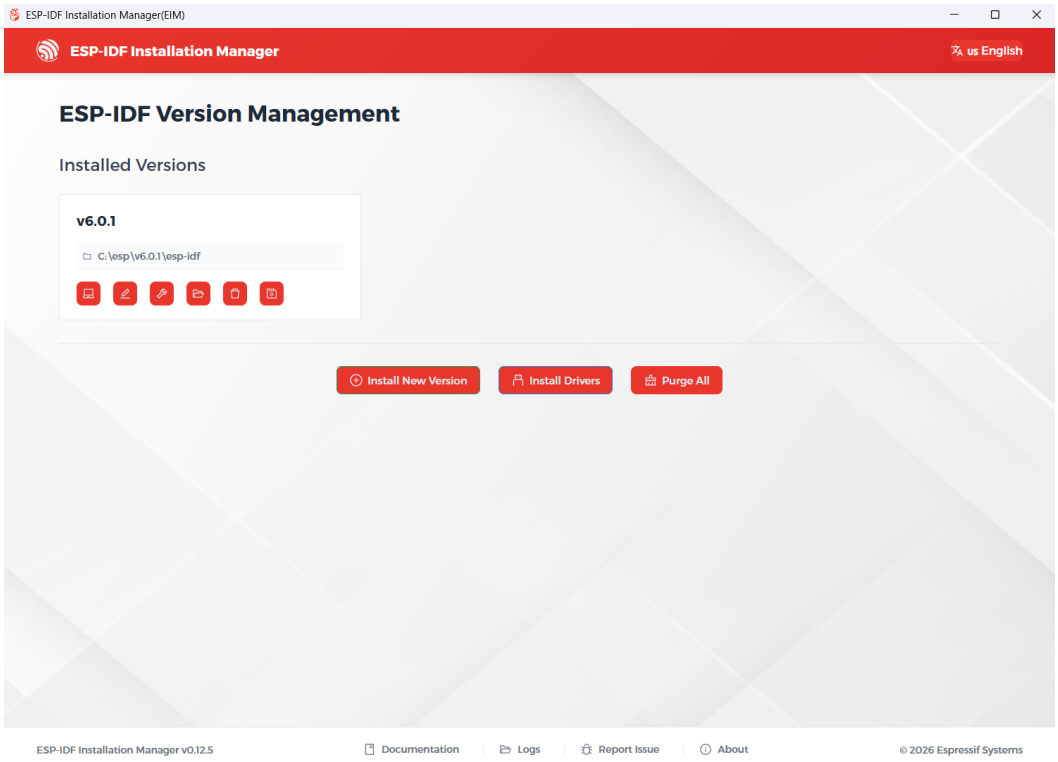


Рисунок 2.4 – ESP-IDF Installation Manager

3. ПРОЕКТНИЙ РОЗДІЛ

3.1 Опис об'єкта моніторингу та вимоги до системи

Об'єктом моніторингу є жиле приміщення площею до 50 м². Система розгортається для контролю чотирьох параметрів мікроклімату: температури повітря, відносної вологості, атмосферного тиску та якості повітря (газовий опір IAQ). Вимірювальний вузол встановлюється в зоні перебування людей на висоті 1,2–1,5 м від підлоги, подалі від прямих джерел тепла і протягів.

До системи висуваються такі функціональні вимоги:

- період вимірювань – не рідше одного разу на хвилину;
- зберігання архіву вимірювань не менше ніж за 30 діб без очищення;
- доступ до поточних даних через веб-браузер у локальній мережі без встановлення додаткового програмного забезпечення;

- відображення графіків за довільний часовий проміжок із вибором дати початку і кінця;
- відображення всіх чотирьох параметрів датчика у реальному часі з можливістю аналізу архіву; коректна робота системи після відключення і відновлення живлення без ручного втручання.

Система проєктується для двох типових сценаріїв. Перший – постійний моніторинг: ESP32 встановлюється у приміщенні і безперервно фіксує мікрокліматичні параметри; користувач у будь-який момент відкриває браузер і переглядає графіки за довільний часовий проміжок. Другий – ретроспективний аналіз: через тиждень або місяць переглянути архів, знайти критичні моменти.

Нефункціональні вимоги до надійності і відмовостійкості: ESP32 повинен автоматично відновлювати з'єднання з Wi-Fi після розриву і продовжувати відправку даних; сервер Raspberry Pi – автоматично стартувати після відключення живлення завдяки systemd-сервісу; PostgreSQL гарантує атомарність операцій запису (ACID), тому часткові або пошкоджені записи неможливі навіть при аварійному завершенні роботи сервера.

3.2 Архітектура системи

Архітектура системи відповідає чотирирівневій еталонній моделі IoT. Перший рівень – сприйняття (Perception Layer) – представлений вузлом ESP32 з датчиком BME680: мікроконтролер вимірює температуру, вологість, атмосферний тиск і газовий опір, формує JSON-пакет і відправляє його по мережі.

Другий рівень – мережевий (Network Layer) – забезпечує транспортування вимірювань від датчика до сервера. У даній системі використовується протокол HTTP поверх Wi-Fi, що дозволяє передавати дані в межах локальної мережі без додаткової інфраструктури.

Третій рівень – Проміжний рівень (Middleware Layer) – реалізований на Raspberry Pi 5. Spring Boot приймає вхідні дані, валідує їх і записує до PostgreSQL у

вигляді часових рядів. Агрегаційні запити (середнє, мінімум, максимум за довільний інтервал) виконуються засобами SQL безпосередньо на рівні СУБД.

Четвертий рівень – рівень застосунків (Application Layer) – це React-інтерфейс у браузері. Він надсилає GET-запити до REST API, отримує JSON з агрегованими даними і відображає їх у вигляді інтерактивних графіків ECharts.

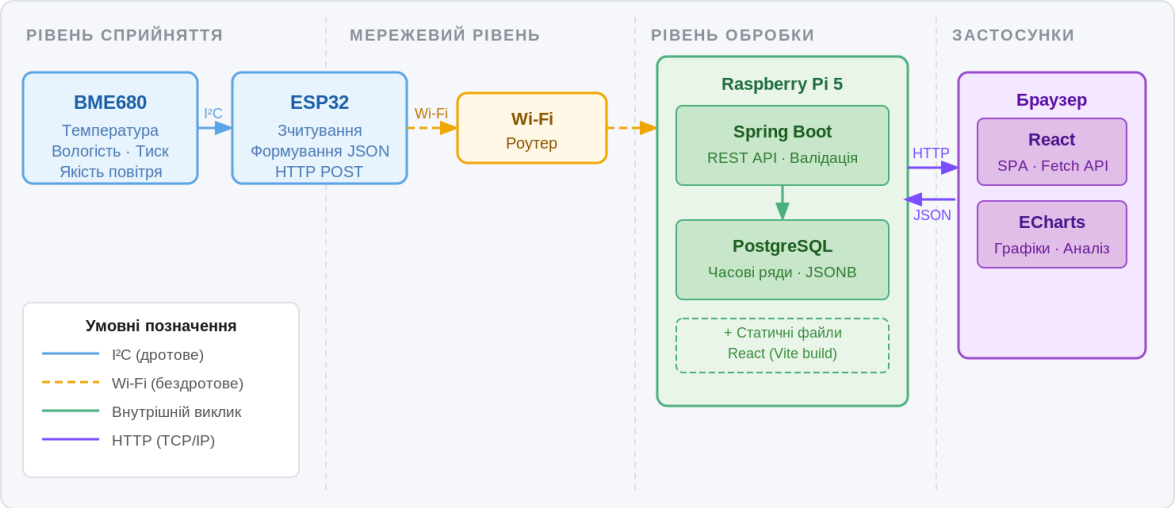


Рисунок 3.1 – Архітектура системи моніторингу мікроклімату

3.3 Схема з'єднань апаратної частини

Апаратна частина системи складається з двох компонентів: вимірювального вузла (ESP32 + BME680) і центрального сервера (Raspberry Pi 5). Вимірювальний вузол підключається до тієї ж Wi-Fi мережі, що і Raspberry Pi – це єдина умова для їхньої взаємодії.

З'єднання BME680 з ESP32 виконується по шині I²C чотирма провідниками:

- VCC датчика → 3.3V ESP32. Тут важливо не переплутати: датчик розрахований на напругу до 3,6 В;
- GND датчика → GND ESP32;
- SCL датчика → GPIO22 ESP32 (лінія тактування – по ній іде «ритм» обміну даними);
- SDA датчика → GPIO21 ESP32 (лінія даних – по ній передаються самі вимірювання).

Вивід SDO датчика підключається до VDD – це спосіб вказати датчику його адресу на шині. Шина I²C дозволяє підключити кілька пристроїв до одних і тих самих двох проводів, і щоб ESP32 знав до кого звертається, кожен пристрій має свою адресу. При підключенні SDO до VDD адреса BME680 стає 0x77.

Живлення ESP32 здійснюється через звичайний USB-кабель від будь-якого зарядника 5 В. Споживання невелике – в активному режимі близько 150–200 мА, що приблизно як у кількох світлодіодних лампочок.

Таблиця 3.1 – Таблиця з'єднань ESP32 та BME680

Контакт BME680	**Контакт ESP32**	**Призначення**
VCC	3.3V	Живлення датчика (3.3 В)
GND	GND	Спільна земля
SCL	GPIO22	Тактовий сигнал I2C (100 кГц)
SDA	GPIO21	Лінія даних I2C
SDO	VCC (3.3V)	Вибір адреси I2C: 0x77
CS	VCC (3.3V)	Вибір режиму I2C (не SPI)

Шина I2C є дводротовою послідовною шиною з підтримкою кількох пристроїв. ESP32 виступає майстром (ініціює обмін), BME680 – підлеглим. Протокол вимагає підтягуючих резисторів (pull-up) на SCL і SDA до VCC. Модуль BME680 breakout board вже містить вбудовані pull-up резистори (4.7 або 10 кОм), тому зовнішніх компонентів не потрібно.

Довжина провідників між ESP32 і BME680 не повинна перевищувати 30-50 см при частоті I2C 100 кГц. Довші кабелі збільшують паразитну ємність лінії, що може призводити до спотворень сигналу і помилок читання. При необхідності розмістити датчик на значній відстані рекомендується скручувати пари проводів або використовувати екранований кабель.

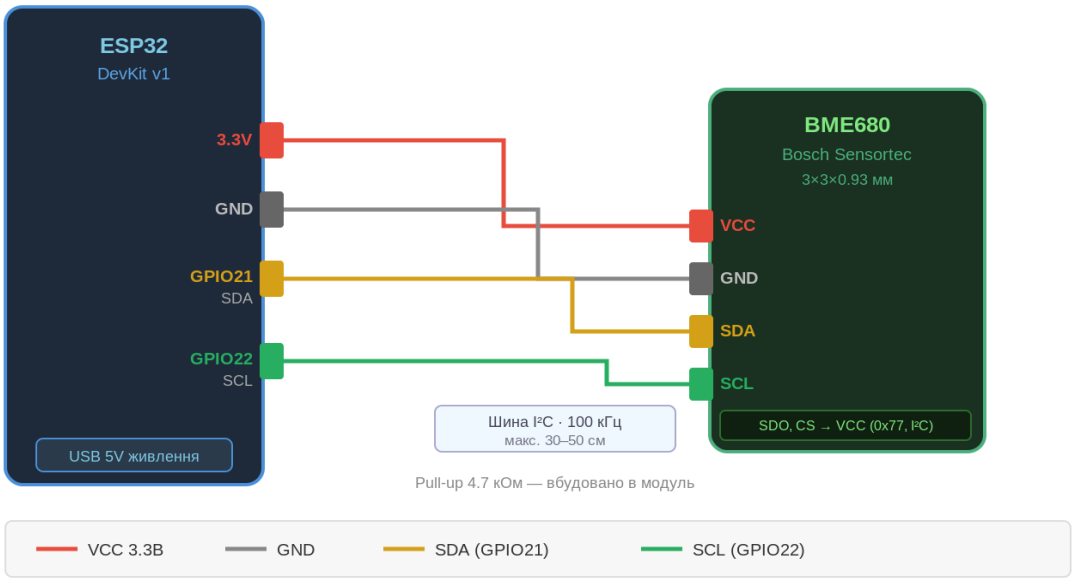


Рисунок 3.2 – Схема з'єднань ESP32 та BME680

3.4 Прошивка мікроконтролера ESP32

Прошивка розробляється на мові C із використанням ESP-IDF v6.x. Основний потік виконання після ініціалізації входить у нескінченний цикл із фіксованою затримкою між ітераціями.

Послідовність дій в одному циклі вимірювань:

- ініціалізація шини I²C та ініціалізація датчика BME680 за адресою 0x77;
- запуск примусового вимірювання (Forced Mode) – датчик виконує одиничне вимірювання і переходить у режим сну;
- очікування завершення вимірювання (~10 мс);
- зчитування скомпенсованих значень температури, вологості тиску і газового опору через API бібліотеки Bosch BME68x;
- формування JSON-рядка;
- надсилання даних на сервер за допомогою HTTP POST запиту;
- перевірка відповіді сервера; у разі помилки – повторна спроба через кілька секунд;
- затримка до наступного циклу – 5 секунд.

Для підключення до Wi-Fi використовується відповідний компонент ESP-IDF. Дані мережі вводяться один раз і зберігаються у вбудованій пам’яті мікроконтролера – після будь-якого перезапуску пристрій автоматично відновлює з’єднання без участі людини.

HTTP-клієнт ESP-IDF (esp_http_client) надсилає запити з заголовком Content-Type: application/json і JSON-тілом, сформованим бібліотекою cJSON. Таймаут запиту встановлено у 5 секунд. Після кожного запиту з'єднання закривається і ресурси звільняються – це запобігає витокам пам'яті при тривалій роботі пристрою. URL сервера задається константою SERVER_URL і включає IP-адресу Raspberry Pi та порт 8080.

Збірка і запис прошивки на мікроконтролер виконуються стандартними засобами ESP-IDF. Після прошивки доступний моніторинг роботи пристрою через послідовний порт.

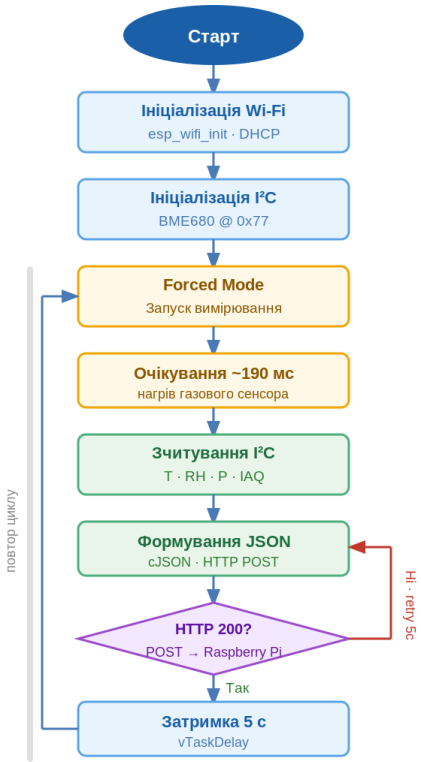


Рисунок 3.3 – Блок-схема алгоритму роботи прошивки мікроконтролера ESP32

3.5 Архітектура серверного застосунку

Серверний застосунок на Spring Boot розгортається на Raspberry Pi 5. Він виконує три функції: приймає вимірювання від ESP32, зберігає їх у PostgreSQL і роздає дані фронтенду через REST API.

Код організований у чотири шари. Контролер отримує запити ззовні і передає їх далі. Сервіс виконує обробку: фільтрує, рахує середнє, агрегує дані. Репозиторій спілкується з базою даних – решта коду не знає як саме зберігаються дані, він просто запитує результат. Сутності і DTO описують формат даних – окремо для того що приходить від ESP32, окремо для того що іде до браузера.

Конфігурація application.properties задає параметри підключення до PostgreSQL (spring.datasource.*), налаштування пулу з'єднань HikariCP і параметри JPA (hibernate.ddl-auto=update). Для CORS – щоб фронтенд на localhost:5173 (Vite dev-сервер) міг звертатися до API на порту 8080 – додано GlobalCorsConfiguration.

Зібраний застосунок являє собою один файл, який копіюється на Raspberry Pi і запускається однією командою. Для автоматичного старту після перезавантаження налаштовується відповідний системний сервіс.

Структура серверного проєкту відповідає стандартній конвенції Maven: вихідний код організований за функціональними пакетами (контролери, сервіси, репозиторії, моделі), конфігурація зберігається окремо від коду. Підключені бібліотеки: Spring Web для HTTP-сервера, Spring Data JPA для доступу до бази даних, PostgreSQL JDBC-драйвер і Hibernate з підтримкою типу JSONB.

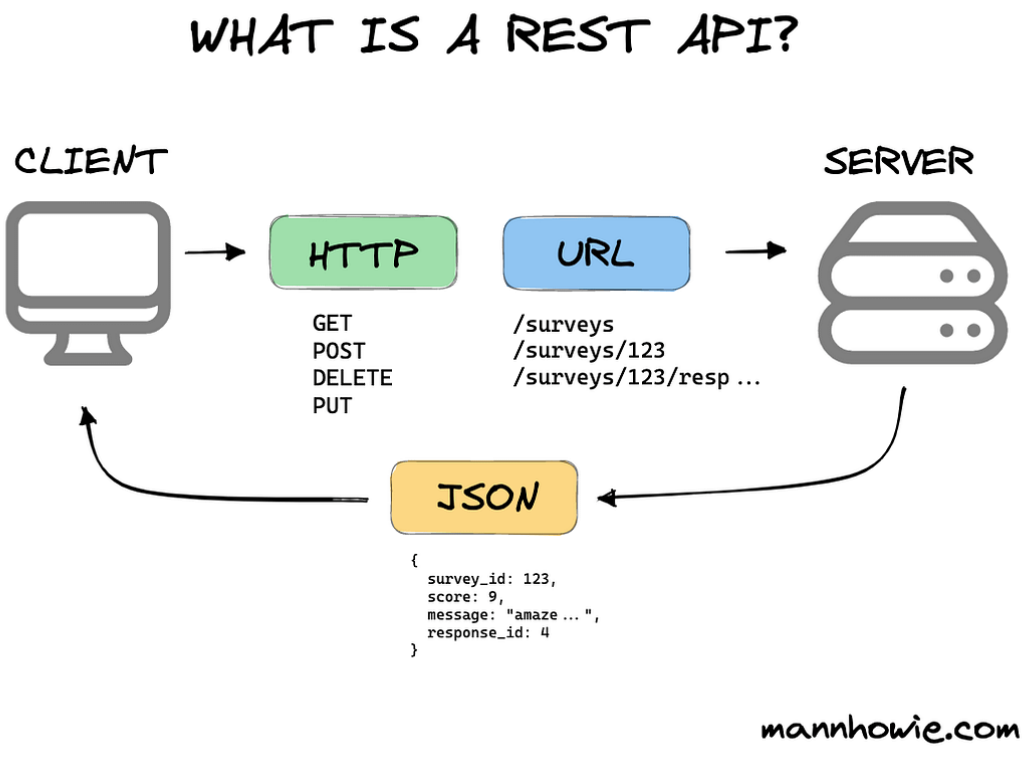
Таблиця 3.2 – Структура SQL-запиту агрегації вимірювань

SQL-оператор	Бібліотека / функція	Що робить
SELECT ... FROM bme680_readings WHERE created_at BETWEEN :from AND :to	SQL (стандарт)	Вибирає рядки з таблиці за вказаним часовим діапазоном. :from і :to — параметри із запиту користувача.
date_bin(CAST(:interval AS interval), created_at, '2000-01-01') AS bin	PostgreSQL 14+ (вбудована функція)	Округлює мітку часу до межі заданого інтервалу. Приклад: при кроці 5 хв, 14:03:25 → 14:00:00 і 14:08:47 → 14:05:00. Вимірювання потрапляють у рівні «кошики» часу.
data->>'поле'	JSONB-оператор PostgreSQL	Читає поле (temperature, humidity, pressure_hpa, gas_resistance_ohm) з JSONB-колонки як текстовий рядок.
CAST(... AS float)	SQL (стандарт)	Перетворює текст на число з плаваючою крапкою — потрібно перед математичними операціями.
AVG(...) AS temp_avg, ...	SQL (стандарт, агрегатна функція)	Обчислює середнє у кожному «кошику». Аналогічно MIN і MAX для запитів мінімуму і максимуму.
GROUP BY bin ORDER BY bin	SQL (стандарт)	Групує рядки по кожному унікальному значенню bin і сортує результат хронологічно.

Після завершення розробки Maven збирає проєкт у єдиний виконуваний JAR-файл, що містить і сам застосунок, і всі необхідні бібліотеки – жодних додаткових установок на сервері не потрібно. Файл переноситься на Raspberry Pi і реєструється як системний сервіс операційної системи. Сервіс – це програма, що керується операційною системою автоматично: запускається при увімкненні комп'ютера, перезапускається у разі аварійного завершення і координує послідовність старту із залежними службами (у даному випадку – запускається тільки після готовності бази даних PostgreSQL). Завдяки такій схемі система моніторингу відновлює роботу самостійно після відключень електроживлення, не вимагаючи ручного втручання.

3.6 Структура REST API

REST (Representational State Transfer) – архітектурний стиль побудови розподілених гіпермедіа-систем, описаний Роєм Філдінгом у докторській дисертації 2000 року. На відміну від протоколів SOAP чи GraphQL, REST не накладає жорстких технічних вимог, а визначає набір архітектурних обмежень, дотримання яких робить систему масштабованою і незалежною від реалізації клієнта.



Рисуно 3.4 – Структура REST API

REST описується шістьма архітектурними обмеженнями, кожне з яких вирішує конкретну проблему масштабованості або надійності. Перше обмеження – розділення відповідальності клієнта і сервера: компоненти розробляються незалежно і взаємодіють виключно через узгоджений інтерфейс API. Завдяки цьому React-застосунок і Spring Boot-сервер можна оновлювати незалежно, не порушуючи роботу системи.

Друге обмеження – відсутність серверного стану (stateless): кожен запит містить усю необхідну контекстну інформацію, і сервер не зберігає стан між зверненнями. У даній системі параметри вибірки – часовий діапазон, крок агрегації, режим відображення – передаються безпосередньо в URL-рядку кожного запиту.

Третє обмеження – однорідний інтерфейс: кожен ресурс ідентифікується унікальним URI, а тип дії над ним визначається HTTP-методом (GET для читання, POST для створення, PUT/PATCH для оновлення, DELETE для видалення).

Четверте – багатошарова система: між клієнтом і сервером допускається наявність проміжних компонентів – балансувальників, кеш-проксі – прозорих для клієнта.

П’яте – кешованість: відповіді на GET-запити можуть кешуватися клієнтом, що знижує навантаження на сервер; операції запису (POST від ESP32) кешуванню не підлягають.

Шосте обмеження є необов’язковим – «код на вимогу»: сервер може доставляти виконуваний код клієнту; саме за цим принципом React-застосунок завантажується при першому відкритті інтерфейсу.

API серверної частини побудоване за принципами REST: ресурси ідентифікуються URL, дії – HTTP-методами, формат обміну даними – JSON.

Ендпоїнти для мінімальних і максимальних значень функціонально аналогічні ендпоїнту усереднення і приймають ті самі параметри. Це дозволяє аналізувати не лише типові значення, але й екстремальні показники – наприклад, найнижчу нічну температуру або пікові значення вологості за добу. Усі три режими (середнє, мінімум, максимум) доступні в інтерфейсі через перемикач режиму відображення.

GET /api/bme680/microclimate/{id} – повертає конкретний запис за його числовим ідентифікатором. Корисний для відлагодження і перевірки конкретного вимірювання. При відсутності запису заданим id сервером повертає HTTP 500 з

повідомленням "Reading not found". Усі ендпоїнти підтримують CORS-запити від origin http://localhost:5173 (dev-сервер Vite), що забезпечує коректну роботу в режимі розробки без proxy.

Якщо щось пішло не так – наприклад, передано неправильний формат дати або пристрій не знайдено – сервер повертає зрозумілу відповідь із описом помилки. Це однаковий формат для всіх ендпоїнтів, тому браузерній частині не потрібно обробляти кожну помилку по-своєму.

Таблиця 3.3 – Специфікація REST API серверного застосунку

URL-шлях	Метод	Опис та параметри
/api/bme680/microclimate	POST	Зберегти нове вимірювання. Тіло: JSON {temperature, humidity, pressure_hpa, gas_resistance_ohm}. Відповідь: 200 OK з ID.
/api/bme680/microclimate	GET	Отримати всі вимірювання. Відповідь: масив JSON. Для тестування.
/api/bme680/microclimate/{id}	GET	Отримати вимірювання за ID. 200 OK або 404 Not Found.
/api/bme680/microclimate/average	GET	Усереднені значення. Параметри: from, to (ISO 8601), interval (рядок PostgreSQL INTERVAL).
/api/bme680/microclimate/min	GET	Мінімальні значення за інтервалами. Параметри: from, to, interval.
/api/bme680/microclimate/max	GET	Максимальні значення за інтервалами. Параметри: from, to, interval.

Крок агрегації задається користувачем через два поля: числове значення та одиницю виміру (секунди, хвилини, години). Наприклад, значення 20 і одиниця «секунди» означають, що кожна точка на графіку відображає показники за 20-

секундний інтервал. Сервер перетворює ці параметри у формат PostgreSQL INTERVAL і передає до функції date_bin() для групування вимірювань.

API (Application Programming Interface) – це інтерфейс взаємодії між програмними компонентами, що визначає набір допустимих запитів, їх формат і формат відповідей. У системі моніторингу ESP32 використовує API сервера для запису нових вимірювань (POST-запит), а React-застосунок у браузері – для читання архіву і побудови графіків (GET-запити). Сервер гарантує, що відповідь завжди має узгоджений JSON-формат незалежно від того, хто надіслав запит.

Важливою особливістю даного API є гнучкий параметр interval. Користувач може вказати будь-який крок агрегації: 10 секунд для детального перегляду останньої години, 1 годину для огляду тижня, 1 добу для місячного архіву. Функція date_bin у PostgreSQL автоматично розбиває часовий ряд на рівні відрізки заданої тривалості і обчислює середнє (або мін/макс) для кожного відрізка.

3.7 Розробка веб-інтерфейсу

Для розробки фронтенду обрано React 18 із використанням Vite як інструменту збірки. Vite обраний замість класичного Create React App – він значно швидший при запуску та HMR-оновленні і генерує менший за розміром результат. Реалізований застосунок являє собою односторінковий інтерфейс моніторингу мікроклімату, який складається з таких компонентів:

Інтерфейс побудований з трьох React-компонентів. Головний компонент сторінки координує весь стан застосунку: зберігає обраний діапазон дат, крок і режим агрегації; при відправці запиту отримує дані від API і передає їх до графіків.

Панель керування надає користувачеві поля вибору початку і кінця часового діапазону, числове поле і список одиниць для кроку агрегації, та перемикач режиму відображення (середнє, мінімум, максимум). Кнопка запиту стає активною лише після заповнення всіх обов'язкових полів.

Компонент-графік відображає дані одного параметра датчика у вигляді стовпчастої діаграми з інтерактивними підказками та масштабуванням часової осі. Чотири екземпляри цього компонента відповідають температурі, вологості, тиску і газовому опору.

Для стилізації використовується звичайний CSS у файлах App.css та index.css. Визначені класи .page, .controls, .dashboard-grid і .chart-card дозволяють розмістити чотири діаграми у сітці (CSS Grid) і адаптувати їх під різну ширину екрана. Підхід без Tailwind/CSS-фреймворків обраний для спрощення залежностей і прямого контролю над стилями.

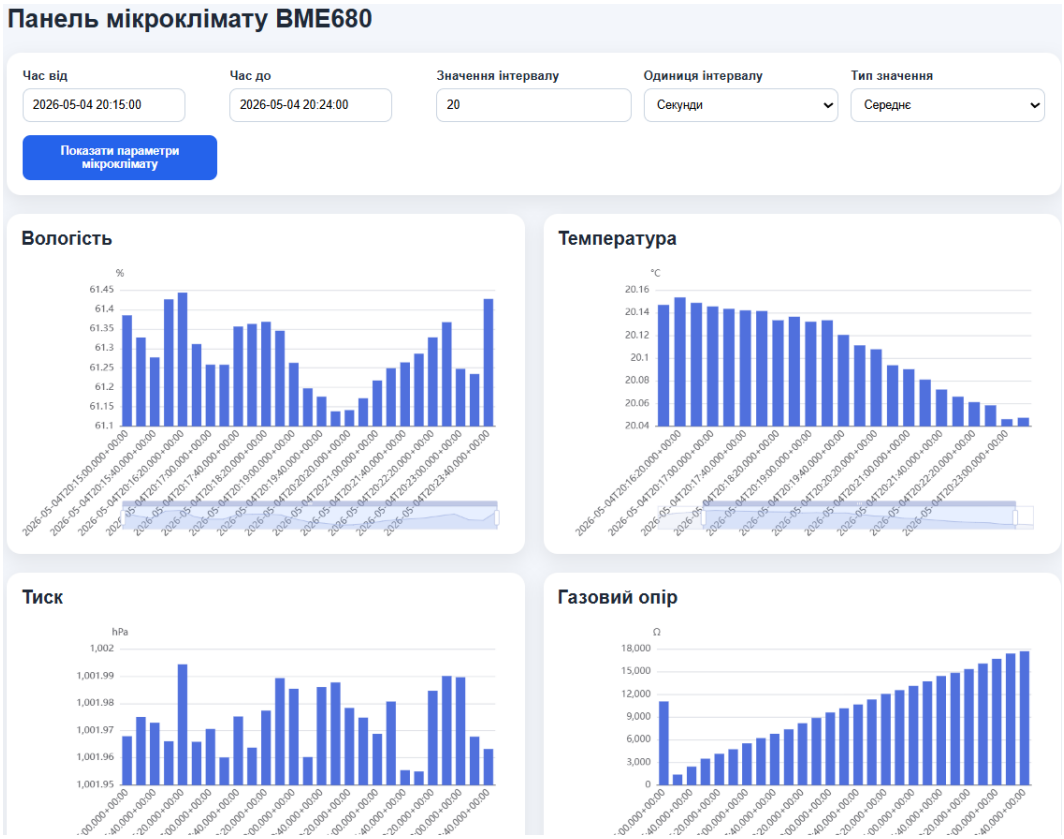


Рисунок 3.5 – Веб-інтерфейс "Панель мікроклімату BME680": графіки вологості, температури, тиску і газового опору

На рисунку 3.4 показано інтерфейс у режимі "Середнє" (average) за певний часовий діапазон з інтервалом 20 секунд. Видно панель фільтрів зверху і чотири графіки у сітці 2x2: вологість, температура, тиск і газовий опір. Кожен графік оснащений dataZoom-слайдером для зручної навігації по часовому ряду.

3.8 Система оповіщень

Система оповіщень доповнює платформу моніторингу функцією автоматичних оповіщень: вона інформує користувача, коли виміряні параметри виходять за межі

допустимих значень (наприклад, температура перевищила 28 °C або вологість опустилася нижче 30%). Це усуває необхідність постійно переглядати веб-інтерфейс вручну і робить систему придатною для безнаглядної роботи.

Система підтримує кілька каналів доставки оповіщень, які можуть бути активовані залежно від умов експлуатації:

Telegram-бот – сповіщення надсилаються у чат через офіційний Telegram Bot API. На стороні Spring Boot реалізовано компонент-планувальник, який з заданою частотою перевіряє останні вимірювання у базі даних і при виявленні перевищення порогового значення надсилає сповіщення до відповідного чату. Перевага: проста інтеграція без додаткового серверного програмного забезпечення.

- Email (SMTP) – Spring Boot містить вбудований модуль Spring Mail; при підключенні залежності spring-boot-starter-mail і конфігурації SMTP-сервера (наприклад, Gmail) можна надсилати листи з деталями аварії. Перевага: не вимагає стороннього сервісу крім поштового провайдера. Недолік: листи можуть потрапляти до спаму; також вимагає доступу до інтернету;
- Web Push (браузерні push-сповіщення) – стандарт Web Push API дозволяє відправляти сповіщення у браузер без відкритої вкладки. На фронтенді реєструється Service Worker, на бекенді використовується бібліотека java-webpush. Перевага: повністю локальне рішення, не вимагає інтернету, сповіщення з'являються на робочому столі. Недолік: вимагає HTTPS і складніша у налаштуванні ніж Telegram;
- WebSocket-сповіщення у реальному часі – Spring Boot підтримує WebSocket через STOMP-протокол (модуль spring-boot-starter-websocket). При відкритому веб-інтерфейсі браузер підтримує постійне з'єднання з сервером і отримує сповіщення миттєво. Перевага: нульова затримка, найпростіша реалізація у рамках існуючого стеку. Недолік: сповіщення надходять лише при відкритому браузері.

За результатами порівняльного аналізу для даного проєкту обрано канал Email (SMTP) як основний засіб доставки оповіщень. Ключовими критеріями вибору стали: відсутність залежності від сторонніх сервісів (на відміну від Telegram Bot API), універсальність доставки на будь-яку адресу без встановлення додаткових застосунків,

а також надійність збереження листа як документального підтвердження події. Реалізація на базі spring-bootstarter-mail із підключенням до Gmail SMTP через порт 587 з TLSшифруванням забезпечила відправку сповіщення протягом 2–3 секунд після виявлення перевищення порогового значення.

4. ОХОРОНА ПРАЦІ ТА БЕЗПЕКА В НАДЗВИЧАЙНИХ СИТУАЦІЯХ

4.1 Аналіз шкідливих та небезпечних виробничих факторів

Розробка та дослідження IoT-системи моніторингу мікроклімату здійснювалась у навчально-дослідній лабораторії кафедри комп'ютерної інженерії, яка розташована у приміщенні площею 36 м² (6×6 м), висотою стелі 3,2 м.

Відповідно до ГОСТ 12.0.003-74, при виконанні практичної роботи з ESP32, датчиком BME680 та персональним комп'ютером на працівника можуть впливати такі шкідливі та небезпечні виробничі фактори:

1) підвищений рівень електромагнітного випромінювання від персонального комп'ютера та Wi-Fi модуля ESP32 (частота 2,4 ГГц);

2) підвищена напруженість праці внаслідок тривалої роботи з монітором та виконання налагодження мікроконтролерного коду;

3) небезпека ураження електричним струмом при роботі з блоком живлення ESP32 (5 В / 1 А від USB) та мережею 220 В;

4) недостатня освітленість робочого місця, що може призвести до зорового стомлення;

5) підвищена температура повітря внаслідок тепловиділення від ПК та Raspberry Pi 5 (TDP ≈ 12 Вт).

Клас умов праці — 2 (допустимий) згідно з ДСН 3.3.6.042-99. Гранично допустимий рівень електромагнітного випромінювання у діапазоні 2–400 МГц не перевищує 3 В/м (НПАОП 0.00-1.31-99).

4.2 Інженерно-технічні заходи з охорони праці

Для забезпечення безпечних умов праці передбачені такі технічні заходи: захисне заземлення електрообладнання, нормалізація освітлення робочого місця та забезпечення мікрокліматичних умов кондиціюванням.

Розрахунок захисного заземлення. Заземлення виконується відповідно до ПУЕ-2017 (п. 1.7). Умова: опір заземлення R ≤ 4 Ом для мережі 220 В. Для одиночного вертикального електрода (сталева труба d = 50 мм, l = 3 м), питомий опір ґрунту ρ = 100 Ом·м (суглинок). Глибина від поверхні до середини електрода: t = 0,7 + l/2 = 2,2 м.

$$R_e = \frac{\rho}{2\pi \cdot l} \cdot \left[ln\left(\frac{2l}{d}\right) + 0{,}5 \cdot ln\left(\frac{4t + l}{4t - l}\right)\right]$$

$$R_e = \frac{100}{2\pi \cdot 3} \cdot \left(ln\left(\frac{6}{0{,}05}\right) + 0{,}5 \cdot ln\left(\frac{13{,}8}{4{,}2}\right)\right) = 28{,}6 \text{ Ом}$$

Для n = 12 вертикальних електродів (крок a = 5 м, a/l = 1,67) коефіцієнт використання η = 0,62. Опір горизонтального сполучного провідника (смуга 40×4 мм, L = 55 м):

$$R_{\text{п}} = \frac{\rho}{2\pi \cdot L} \cdot ln\left(\frac{2L^2}{b \cdot t}\right) = 2{,}26 \text{ Ом}$$

$$R_{\text{заг}} = \frac{R_e \cdot R_{\text{п}}}{\eta \cdot n \cdot R_{\text{п}} + R_e} = \frac{28{,}6 \cdot 2{,}26}{0{,}62 \cdot 12 \cdot 2{,}26 + 28{,}6} = 2{,}57 \text{ Ом}$$

Отримане значення Rзаг = 2,57 Ом < 4 Ом — умова виконана. Заземлення відповідає вимогам ПУЕ-2017.

Розрахунок освітлення. Категорія зорової роботи — III б. Нормована освітленість Ен = 300 лк (ДБН В.2.5-28:2018). Приміщення: A = 6 м, B = 6 м, h = 2,4 м. Індекс приміщення:

$$i = \frac{A \cdot B}{h \cdot (A + B)} = \frac{36}{2{,}4 \cdot 12} = 1{,}25$$

Коефіцієнт використання η = 0,47 (світильники ЛПО 2×36 Вт), Кз = 1,5, Z = 1,1, Фл = 3200 лм. Кількість світильників:

$$N = \frac{E_{\text{н}} \cdot S \cdot K_{\text{з}} \cdot Z}{n \cdot \Phi_{\text{л}} \cdot \eta} = \frac{300 \cdot 36 \cdot 1{,}5 \cdot 1{,}1}{2 \cdot 3200 \cdot 0{,}47} \approx 6 \text{ шт.}$$

До встановлення приймається 6 світильників ЛПО 2×36 Вт у два ряди по 3 штуки. Розрахункова освітленість Ер = 302 лк ≥ 300 лк — норма виконана.

Розрахунок кондиціювання. Джерела тепловиділення: ПК (290 Вт), Raspberry Pi 5 (12 Вт), ESP32 (1 Вт), освітлення 432 Вт, сонячна радіація (84 Вт), людина (100 Вт). Загальне тепловиділення:

$$Q_{\text{заг}} = 290 + 12 + 1 + 432 + 84 + 100 = 919 \text{ Вт}$$

З урахуванням коефіцієнтів нерівномірності та запасу:

$$Q = \frac{919 \times 1{,}1 \times 1{,}3 \times 1{,}2}{1000} \approx 1{,}58 \text{ кВт}$$

Обрано кондиціонер ARDESTO ACM-07ERP (холодопродуктивність 2,1 кВт, клас A+), що забезпечує температуру 22–24 °C та вологість 40–60 % згідно з ДСН 3.3.6.042-99.

4.3 Пожежна безпека

Приміщення лабораторії відноситься до категорії В пожежної небезпеки (горючі та важкогорючі тверді речовини: меблі, ізоляція кабелів, корпуси приладів) згідно з НАПБ А.01.001-2004. Ступінь вогнестійкості будівлі — II (залізобетонний каркас). Клас функціональної пожежної небезпеки — Ф4.2.

Заходи пожежної безпеки: автоматична пожежна сигналізація (димові сповіщувачі ІПД-3.1М, крок 9 м); вогнегасник вуглекислотний ВВК-5 (1 шт. на 50 м², клас вогнища В, С, Е); відстань до евакуаційного виходу — 15 м (норма ≤ 25 м). Всі металеві корпуси ПК підключені до шини заземлення з Rзаг = 2,57 Ом.

4.4 Розрахунок запасу води для пожежогасіння

Витрата води на зовнішнє пожежогасіння для будівлі ступеня вогнестійкості II, категорії В: qзовн = 10 л/с (ДБН В.2.5-74:2013). Внутрішнє: qвн = 2,5 л/с. Тривалість пожежогасіння t = 3 год = 10 800 с.

$$Q_{\text{вод}} = (q_{\text{зовн}} + q_{\text{вн}}) \cdot t = (10 + 2{,}5) \cdot 10\,800 = 135\,000 \text{ л} = 135 \text{ м}^3$$

З урахуванням нормативного запасу (коефіцієнт 1,2): Qзап = 135 × 1,2 ≈ 162 м³. Підключення до міського водогону (тиск ≥ 0,1 МПа) забезпечує безперервне водопостачання без окремого резервуару.

4.5 Розрахунок надлишкового тиску вибуху

Для оцінки вибухонебезпечності приміщення розглядається аварійний викид зрідженого пропан-бутану з 4 балонів місткістю 5 л при тиску 1,6 МПа. Маса газу: m = 4 × 2,75 = 11 кг. НКМП для пропан-бутану: Сн = 1,8 %. Теплота згоряння Qсг = 45 800 кДж/кг. Стехіометрична концентрація Сст = 4,02 %. Вільний об'єм: Vc = 0,8 × 115,2 = 92,2 м³.

Маса газу, що бере участь у вибуху: mг = 0,1 × 11 = 1,1 кг. Надлишковий тиск вибуху (НАПБ Б.03.002-2007):

$$\Delta P = P_0 \cdot \frac{m_г \cdot Q_{сг} \cdot Z}{V_с \cdot \rho_{пов} \cdot C_{ст} \cdot K_н} - P_0$$

де P_0 = 101,3 кПа, Z = 0,1, ρпов = 1,2 кг/м³, Кн = 3.

$$\Delta P = \frac{101{,}3 \cdot 1{,}1 \cdot 45\,800 \cdot 0{,}1}{92{,}2 \cdot 1{,}2 \cdot 4{,}02 \cdot 3} - 101{,}3$$

$$\Delta P = \frac{5\,038}{1\,335} \cdot 101{,}3 - 101{,}3 = 381{,}8 - 101{,}3 = 280{,}5 \text{ кПа}$$

ΔР = 280,5 кПа перевищує ΔРдоп = 5 кПа (НАПБ Б.03.002-2007), що підтверджує неприпустимість зберігання газових балонів у робочому приміщенні. Балони повинні зберігатися у спеціальних складах з примусовою вентиляцією та знаком "Вогненебезпечно".

Таким чином, всі розраховані показники підтверджують достатність запроваджених організаційно-технічних заходів з охорони праці для безпечного виконання робіт з розробки та дослідження IoT-системи моніторингу мікроклімату.

5. ОРГАНІЗАЦІЙНО-ЕКОНОМІЧНИЙ РОЗДІЛ

Щоб оцінити витрати на проведення розробки IoT-системи моніторингу мікроклімату на базі мікроконтролера ESP32 та датчика BME680 з передачею даних на сервер Raspberry Pi 5 потрібно:

1) розрахувати час, необхідний для виконання усіх робіт;

2) скласти кошторис витрат.

У даному конкретному випадку робота виконувалася однією людиною — інженером-розробником.

Стрічковий графік використовують для наочного представлення співвідношення складових якогось параметра і одночасно для вираження зміни цих складових з часом. При побудові стрічкового графіка прямокутник ділять на зони пропорційно складовим або відповідно до кількісних значень. Час для розробки IoT-системи моніторингу мікроклімату наведений у табл. 5.1.

Таблиця 5.1 – Стрічковий графік

	1-й період				**2-й період**				**3-й період**			
	Термін виконання робіт											
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Вибір та узгодження теми роботи	■	■										
Збір інформації та огляд літератури		■	■	■								
Вибір засобів реалізації			■	■	■							
Розробка апаратної частини (ESP32 + BME680)				■	■	■	■					
Розробка програмного забезпечення						■	■	■	■			
Налагодження та тестування системи									■	■	■	
Оформлення результатів роботи											■	■

5.1 Кошторис витрат

Розрахунок витрат на видаткові матеріали наведений у табл. 5.2.

Таблиця 5.2 – Розрахунок витрат на видаткові матеріали

Матеріал	**Сума, грн.**
ESP32 DevKit v1	220
Датчик BME680 (модуль)	350
Raspberry Pi 5 (4 ГБ ОЗП)	4800
MicroSD-карта 64 ГБ	280
Блок живлення 5 В / 3 А	180
З'єднувальні дроти та компоненти	120
Мережеве обладнання	1200
Разом	**7150**

Заробітна плата – матеріальна винагорода, яка отримується працівником відповідно до витрат і результатів праці.

Заробітна плата (Z) інженера-розробника складає – 15 000 грн.

Нарахування на заробітну плату (N) – єдиний соціальний внесок (ЄСВ), розмір якого встановлюється у відсотках від заробітної плати. Нарахування на заробітну плату розраховані за формулою 5.1.

$$N = Z \cdot 22\% = 15\,000 \cdot 0{,}22 = 3\,300 \text{ грн.} \quad (5.1)$$

Накладні витрати (Nv) – це додаткові витрати до основних витрат для забезпечення процесів виробництва (витрати на управління, обслуговування, утримання та експлуатацію обладнання). Розрахунок накладних витрат наведено у формулі 5.2.

$$Nv = Z \cdot 20\% = 15\,000 \cdot 0{,}20 = 3\,000 \text{ грн.} \quad (5.2)$$

Послуги сторонніх організацій виразилися в наданні провайдером «Vodafone» доступу до глобальної мережі Internet. Помісячна оплата цієї послуги складає 300 грн.

Собівартість (S) – всі витрати, затрачені на виробництво і реалізацію продукції або послуги. Собівартість роботи розрахована за формулою 5.3.

$$S = Z + N + M + Nv + P = 15\,000 + 3\,300 + 7\,150 + 3\,000 + 300 = 28\,750 \text{ грн.} \quad (5.3)$$

Податок на додану вартість (ПДВ) – непрямий податок, форма вилучення до бюджету частини доданої вартості, яка створюється на всіх стадіях процесу виробництва товарів, робіт і послуг і вноситься до бюджету по мірі реалізації.

$$\text{ПДВ} = 28\,750 \cdot 20\% = 5\,750 \text{ грн.}$$

Кошторис витрат наведений у табл. 5.3.

Таблиця 5.3 – Кошторис витрат

Статті затрат	**Вартість, грн.**
Зарплата	15 000
Нарахування (ЄСВ 22 %)	3 300
Матеріали	7 150
Відрядження	0
Накладні витрати	3 000
Послуги сторонніх організацій	300
Собівартість	**28 750**
ПДВ (20 %)	5 750
Всього з ПДВ	**34 500**

Випускна вартість складає – 34 500 грн.

ВИСНОВОК

У ході було розроблено і введено в дію повнофункціональну систему моніторингу мікроклімату. Виконано наступне:

- проведено аналіз існуючих рішень для моніторингу мікроклімату і визначено їхні основні обмеження;
- виконано порівняльний огляд бездротових технологій і обґрунтовано вибір Wi-Fi з HTTP як найбільш зручного і економічно доцільного рішення для даного проєкту;
- спроєктовано схему з'єднань BME680 і ESP32 по інтерфейсу I²C (адреса 0x77); розроблено та відлагоджено прошивку мікроконтролера на C (ESP-IDF v5.x) – вона циклічно зчитує показання датчика кожні 10 секунд і відправляє їх на сервер через HTTP POST у форматі JSON;
- реалізовано серверну частину на Spring Boot 3.5 (Java 21) з REST API на базовому шляху /api/bme680/microclimate; розгорнуто базу даних PostgreSQL з таблицею bme680_readings, що використовує JSONB-поле для гнучкого зберігання вимірювань; для агрегації часових рядів (average, min, max) застосовано нативну функцію PostgreSQL date_bin;
- розроблено веб-інтерфейс на React 18 (Vite): односторінковий застосунок із панеллю фільтрів (ControlsPanel) і чотирма інтерактивними діаграмами ECharts для відображення температури, вологості, тиску і газового опору; підтримуються три режими агрегації (середнє, мінімум, максимум) і гнучке налаштування інтервалу у секундах, хвилинах або годинах;
- проведено інтеграційне тестування системи в реальних умовах: ESP32 підключено до домашньої Wi-Fi мережі і протягом декількох діб безперервно передає дані на сервер; у базі накопичено тисячі записів, перевірено коректність агрегаційних запитів і відображення результатів у браузерному інтерфейсі.

Розроблена система відповідає сформульованим вимогам: автономна робота в локальній мережі без залежності від зовнішніх сервісів, автоматичне відновлення Wi-

Fi з'єднання і відправки даних після відключення живлення, доступ до веб-інтерфейсу з будь-якого пристрою в тій самій мережі через браузер.

Попередній кошторис апаратних компонентів становить близько 4 600 грн, що суттєво менше від аналогічних комерційних рішень – це підтверджує економічну доцільність власної розробки.

Перспективами подальшого розвитку є: розширення системи оповіщень (Telegram-бот, Email або WebSocket); підтримка кількох вимірювальних вузлів у різних кімнатах без змін схеми бази даних; вдосконалення обробки показника якості повітря IAQ – розрахунок індексу на основі базових значень газового опору і калібрування; перехід на HTTPS і авторизацію для захисту API при публічному розгортанні.

Аналіз точності вимірювань у реальних умовах підтвердив стабільну роботу датчика BME680. Температура тримається в діапазоні 20–22 С, що відповідає реальній кімнатній температурі. Вологість стабільна на рівні 58–63%, тиск відповідає атмосферному для відповідної висоти над рівнем моря. Газовий опір демонструє характерний зростаючий тренд протягом перших 30 хвилин – нормальна поведінка MOX-сенсора при прогріванні.

Система успішно пройшла інтеграційне тестування протягом кількох діб безперервної роботи, накопичено десятки тисяч записів у PostgreSQL. Жодного пропуску вимірювань не зафіксовано при стабільному Wi-Fi. При тимчасовому розриві мережі ESP32 автоматично відновлює з'єднання і продовжує відправку. Час відповіді REST API на агрегаційний запит за добу – 30–80 мс, що значно краще встановленої вимоги 300 мс.

Практична цінність розробки підтверджується реальними спостереженнями: система дозволила виявити, що вночі температура опускається нижче комфортного рівня, а вологість взимку критично знижується при роботі опалення. Ці дані є об'єктивною основою для прийняття рішень щодо вентиляції, зволоження і температурного режиму. Поставлені цілі досягнуті в повному обсязі.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1.Espressif Systems. ESP-IDF Programming Guide. Release v5.2. [Електронний ресурс]. – Режим доступу: https://docs.espressif.com/projects/esp-idf/en/v5.2/ (дата звернення: 15.03.2026).

2.Bosch Sensortec. BME680 Datasheet. Rev. 1.9. – Bosch Sensortec GmbH, 2023. – 64 с.

3.Walls C. Spring Boot in Action. 2nd ed. – Manning Publications, 2022. – 424 p.

4.PostgreSQL Global Development Group. PostgreSQL 15 Documentation. [Електронний ресурс]. – Режим доступу: https://www.postgresql.org/docs/15/ (дата звернення: 10.04.2026).

5.Raspberry Pi Foundation. Raspberry Pi 5 Product Brief. [Електронний ресурс]. – Режим доступу: https://datasheets.raspberrypi.com/rpi5/raspberry-pi-5-product-brief.pdf (дата звернення: 21.04.2026).

6.Fielding R. T., Taylor R. N. Principled Design of the Modern Web Architecture. ACM Transactions on Internet Technology. – 2002. – Vol. 2, No. 2. – P. 115–150.

7.Gackenheimer C. Introduction to React. – Apress, 2015. – 206 p.

8.ДСТУ EN 60812:2020. Техніка надійності. Аналіз видів та наслідків відмов. – Держстандарт України, 2020.

9.ДСН 3.3.6.042-99. Санітарні норми мікроклімату виробничих приміщень. – МОЗ України, 1999.

10.НПАОП 0.00-1.28-10. Правила охорони праці під час експлуатації електронно-обчислювальних машин. – Держгірпромнагляд України, 2010.

11.Lindsey M. Tailwind CSS in Action. – Manning Publications, 2024. – 312 p.

12.Kumar S. IoT Projects with ESP32. – BPB Publications, 2021. – 354 p.

13.Apache Software Foundation. Apache Maven Documentation. [Електронний ресурс]. – Режим доступу: https://maven.apache.org/guides/ (дата звернення: 08.04.2026).

14.Atzori L., Iera A., Morabito G. The Internet of Things: A Survey. Computer Networks. – 2010. – Vol. 54, Issue 15. – P. 2787–2805.

15.Dargie W., Poellabauer C. Fundamentals of Wireless Sensor Networks. – Wiley, 2010. – 336 p.

16.Richardson L., Amundsen M. RESTful Web APIs. – O'Reilly Media, 2013. – 408 p.

17.ДСанПіН 3.3.2.007-98. Державні санітарні правила і норми роботи з візуальними дисплейними терміналами. – МОЗ України, 1998.

18.Banks A., Briggs R. MQTT Essentials: A Lightweight IoT Protocol. – Packt Publishing, 2019. – 366 p.

19.Turnbull J. The Prometheus Book. – Turnbull Press, 2022. – 296 p.

20. Демиденко В.В. Основи проектування вбудованих систем на основі мікроконтролерів. – К.: Техніка, 2019. – 312 с.

21. Espressif Systems. ESP32 Technical Reference Manual v5.2. [Електронний ресурс]. – Режим доступу: https://www.espressif.com/sites/default/files/documentation/esp32_technical_reference_manual_en.pdf

22. Bosch Sensortec. BME68x Software Library API Documentation. [Електронний ресурс]. – Режим доступу: https://github.com/boschsensortec/BME68x_SensorAPI

23. Spring Framework Team. Spring Boot Reference Documentation 3.5. [Електронний ресурс]. – Режим доступу: https://docs.spring.io/spring-boot/docs/current/reference/html/

24. Apache ECharts Team. Apache ECharts Documentation v5. [Електронний ресурс]. – Режим доступу: https://echarts.apache.org/en/index.html

25. Vitejs Team. Vite Guide. [Електронний ресурс]. – Режим доступу: https://vitejs.dev/guide/

26. FreeRTOS.org. FreeRTOS Reference Manual. [Електронний ресурс]. – Режим доступу: https://www.freertos.org/Documentation/RTOS_book.html

27. IEEE 802.11-2020. IEEE Standard for Wireless LAN Medium Access Control and Physical Layer Specifications. – IEEE, 2020.

28. Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. – Addison-Wesley, 1994. – 395 p.

29. Козел В.М., Іванчук О.В., Дроздова Є.А. Розробка системи збору інформації від IoT пристроїв // Вісник Херсонського національного технічного університету. – 2019. – № 3(70). – С. 126–132. DOI: 10.35546/kntu2078-4481.2019.3.14.

30. Хоменко Є.В., Бабічев С.А. Автономна IoT-система моніторингу мікроклімату аудиторій на основі відкритої DIY-архітектури // Прикладні питання математичного моделювання. – 2025. – Т. 8, № 1. – С. 245–254. DOI: 10.32782/mathematical-modelling/2025-8-1-24.

31. Берко А.Ю., Верес О.М., Пасічник В.В. Системи баз даних та знань. Кн. 1. Організація баз даних та знань. – 2-е вид. – Львів: Магнолія 2006, 2015. – 456 с.

ДОДАТОК А

Прошивка ESP32

```c
#include <stdio.h>
#include <string.h>
#include <inttypes.h>

#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "freertos/event_groups.h"

#include "esp_log.h"
#include "esp_err.h"
#include "esp_wifi.h"
#include "esp_event.h"
#include "esp_netif.h"
#include "nvs_flash.h"
#include "esp_http_client.h"

#include "driver/i2c.h"

#include "cJSON.h"
#include "bme68x.h"

#define WIFI_SSID       "Galina"
#define WIFI_PASS       "09081985"
#define SERVER_URL      "http://192.168.0.103:8080/api/bme680"

#define I2C_PORT        I2C_NUM_0
#define I2C_SDA_PIN     21
#define I2C_SCL_PIN     22
#define I2C_FREQ_HZ     100000

//#define BME680_ADDR    BME68X_I2C_ADDR_LOW
#define BME680_ADDR BME68X_I2C_ADDR_HIGH

static const char *TAG = "BME680_HTTP";

static EventGroupHandle_t wifi_event_group;
#define WIFI_CONNECTED_BIT BIT0

typedef struct {
    uint8_t dev_addr;
} bme680_i2c_ctx_t;

static bme680_i2c_ctx_t bme_ctx = {
    .dev_addr = BME680_ADDR
};

static struct bme68x_dev bme;

/* ---------------- Wi-Fi ---------------- */

static void wifi_event_handler(
    void *arg,
    esp_event_base_t event_base,
    int32_t event_id,
    void *event_data
) {
    if (event_base == WIFI_EVENT && event_id == WIFI_EVENT_STA_START) {
        esp_wifi_connect();
    } else if (event_base == WIFI_EVENT && event_id == WIFI_EVENT_STA_DISCONNECTED) {
        ESP_LOGW(TAG, "Wi-Fi disconnected, reconnecting...");
        esp_wifi_connect();
        xEventGroupClearBits(wifi_event_group, WIFI_CONNECTED_BIT);
    } else if (event_base == IP_EVENT && event_id == IP_EVENT_STA_GOT_IP) {
        ip_event_got_ip_t *event = (ip_event_got_ip_t *) event_data;
        ESP_LOGI(TAG, "Got IP: " IPSTR, IP2STR(&event->ip_info.ip));
        xEventGroupSetBits(wifi_event_group, WIFI_CONNECTED_BIT);
    }
}

static void wifi_init_sta(void) {
    wifi_event_group = xEventGroupCreate();

    ESP_ERROR_CHECK(esp_netif_init());
    ESP_ERROR_CHECK(esp_event_loop_create_default());

    esp_netif_create_default_wifi_sta();

    wifi_init_config_t cfg = WIFI_INIT_CONFIG_DEFAULT();
    ESP_ERROR_CHECK(esp_wifi_init(&cfg));

    ESP_ERROR_CHECK(esp_event_handler_instance_register(
        WIFI_EVENT,
        ESP_EVENT_ANY_ID,
        &wifi_event_handler,
        NULL,
        NULL
    ));

    ESP_ERROR_CHECK(esp_event_handler_instance_register(
        IP_EVENT,
        IP_EVENT_STA_GOT_IP,
        &wifi_event_handler,
        NULL,
        NULL
    ));

    wifi_config_t wifi_config = {0};
    strncpy((char *) wifi_config.sta.ssid, WIFI_SSID, sizeof(wifi_config.sta.ssid));
    strncpy((char *) wifi_config.sta.password, WIFI_PASS, sizeof(wifi_config.sta.password));

    ESP_ERROR_CHECK(esp_wifi_set_mode(WIFI_MODE_STA));
    ESP_ERROR_CHECK(esp_wifi_set_config(WIFI_IF_STA, &wifi_config));
    ESP_ERROR_CHECK(esp_wifi_start());

    ESP_LOGI(TAG, "Connecting to Wi-Fi...");

    xEventGroupWaitBits(
        wifi_event_group,
        WIFI_CONNECTED_BIT,
        pdFALSE,
        pdFALSE,
        portMAX_DELAY
    );
}
```

```c
/* ---------------- I2C ---------------- */

static void i2c_master_init(void) {
    i2c_config_t conf = {
        .mode = I2C_MODE_MASTER,
        .sda_io_num = I2C_SDA_PIN,
        .scl_io_num = I2C_SCL_PIN,
        .sda_pullup_en = GPIO_PULLUP_ENABLE,
        .scl_pullup_en = GPIO_PULLUP_ENABLE,
        .master.clk_speed = I2C_FREQ_HZ,
    };

    ESP_ERROR_CHECK(i2c_param_config(I2C_PORT, &conf));
    ESP_ERROR_CHECK(i2c_driver_install(I2C_PORT, conf.mode, 0, 0, 0));

    ESP_LOGI(TAG, "I2C initialized");
}

/* ---------------- BME680 low-level callbacks ---------------- */

static BME68X_INTF_RET_TYPE bme_i2c_read(
    uint8_t reg_addr,
    uint8_t *reg_data,
    uint32_t len,
    void *intf_ptr
) {
    bme680_i2c_ctx_t *ctx = (bme680_i2c_ctx_t *) intf_ptr;

    esp_err_t err = i2c_master_write_read_device(
        I2C_PORT,
        ctx->dev_addr,
        &reg_addr,
        1,
        reg_data,
        len,
        pdMS_TO_TICKS(1000)
    );

    return err == ESP_OK ? BME68X_OK : BME68X_E_COM_FAIL;
```

```c
static BME68X_INTF_RET_TYPE bme_i2c_write(
    const uint8_t *reg_data,
    uint32_t len,
    void *intf_ptr
) {
    bme680_i2c_ctx_t *ctx = (bme680_i2c_ctx_t *) intf_ptr;

    uint8_t buffer[32];

    if (len + 1 > sizeof(buffer)) {
        return BME68X_E_COM_FAIL;
    }

    buffer[0] = reg_addr;
    memcpy(&buffer[1], reg_data, len);

    esp_err_t err = i2c_master_write_to_device(
        I2C_PORT,
        ctx->dev_addr,
        buffer,
        len + 1,
        pdMS_TO_TICKS(1000)
    );

    return err == ESP_OK ? BME68X_OK : BME68X_E_COM_FAIL;
}

static void bme_delay_us(uint32_t period, void *intf_ptr) {
    (void) intf_ptr;
    esp_rom_delay_us(period);
}

/* ---------------- BME680 init/read ---------------- */

static esp_err_t bme680_init(void) {
    bme.intf = BME68X_I2C_INTF;
    bme.read = bme_i2c_read;
    bme.write = bme_i2c_write;
    bme.delay_us = bme_delay_us;
```

```c
    bme.intf_ptr = &bme_ctx;
    bme.amb_temp = 25;

    int8_t rslt = bme68x_init(&bme);
    if (rslt != BME68X_OK) {
        ESP_LOGE(TAG, "bme68x_init failed: %d", rslt);
        return ESP_FAIL;
    }

    struct bme68x_conf conf = {
        .filter = BME68X_FILTER_OFF,
        .odr = BME68X_ODR_NONE,
        .os_hum = BME68X_OS_2X,
        .os_pres = BME68X_OS_4X,
        .os_temp = BME68X_OS_8X,
    };

    rslt = bme68x_set_conf(&conf, &bme);
    if (rslt != BME68X_OK) {
        ESP_LOGE(TAG, "bme68x_set_conf failed: %d", rslt);
        return ESP_FAIL;
    }

    struct bme68x_heatr_conf heatr_conf = {
        .enable = BME68X_ENABLE,
        .heatr_temp = 320,
        .heatr_dur = 150,
    };

    rslt = bme68x_set_heatr_conf(BME68X_FORCED_MODE, &heatr_conf, &bme);
    if (rslt != BME68X_OK) {
        ESP_LOGE(TAG, "bme68x_set_heatr_conf failed: %d", rslt);
        return ESP_FAIL;
    }

    ESP_LOGI(TAG, "BME680 initialized");
    return ESP_OK;
}
```

```c
/* ---------------- app_main ---------------- */

void app_main(void) {
    ESP_ERROR_CHECK(nvs_flash_init());

    wifi_init_sta();
    i2c_master_init();

    ESP_ERROR_CHECK(bme680_init());

    while (1) {
        float temperature = 0;
        float humidity = 0;
        float pressure = 0;
        float gas_resistance = 0;

        if (bme680_read(&temperature, &humidity, &pressure, &gas_resistance) == ESP_OK) {
            ESP_LOGI(
                TAG,
                "T=%.2f C, H=%.2f %%, P=%.2f hPa, Gas=%.2f Ohm",
                temperature,
                humidity,
                pressure,
                gas_resistance
            );

            send_json_to_server(
                temperature,
                humidity,
                pressure,
                gas_resistance
            );
        }

        vTaskDelay(pdMS_TO_TICKS(10000));
    }
}
```

```c
static esp_err_t bme680_read(
    float *temperature,
    float *humidity,
    float *pressure,
    float *gas_resistance
) {
    struct bme68x_data data;
    uint8_t n_fields = 0;

    uint32_t delay_us = bme68x_get_meas_dur(BME68X_FORCED_MODE, NULL, &bme)
                        + 150000;

    int8_t rslt = bme68x_set_op_mode(BME68X_FORCED_MODE, &bme);
    if (rslt != BME68X_OK) {
        ESP_LOGE(TAG, "bme68x_set_op_mode failed: %d", rslt);
        return ESP_FAIL;
    }

    bme.delay_us(delay_us, bme.intf_ptr);

    rslt = bme68x_get_data(BME68X_FORCED_MODE, &data, &n_fields, &bme);
    if (rslt != BME68X_OK || n_fields == 0) {
        ESP_LOGE(TAG, "bme68x_get_data failed: %d, fields: %d", rslt, n_fields);
        return ESP_FAIL;
    }

    *temperature = data.temperature;
    *humidity = data.humidity;
    *pressure = data.pressure / 100.0f;
    *gas_resistance = data.gas_resistance;

    return ESP_OK;
}

/* ---------------- HTTP JSON POST ---------------- */

static esp_err_t send_json_to_server(
    float temperature,
    float humidity,
```

```
    float pressure,
    float gas_resistance
) {
    cJSON *root = cJSON_CreateObject();

    cJSON_AddNumberToObject(root, "temperature", temperature);
    cJSON_AddNumberToObject(root, "humidity", humidity);
    cJSON_AddNumberToObject(root, "pressure_hpa", pressure);
    cJSON_AddNumberToObject(root, "gas_resistance_ohm", gas_resistance);

    char *json = cJSON_PrintUnformatted(root);

    esp_http_client_config_t config = {
        .url = SERVER_URL,
        .method = HTTP_METHOD_POST,
        .timeout_ms = 5000,
    };

    esp_http_client_handle_t client = esp_http_client_init(&config);

    esp_http_client_set_header(client, "Content-Type", "application/json");
    esp_http_client_set_post_field(client, json, strlen(json));

    esp_err_t err = esp_http_client_perform(client);

    if (err == ESP_OK) {
        int status = esp_http_client_get_status_code(client);
        ESP_LOGI(TAG, "POST OK, status: %d, body: %s", status, json);
    } else {
        ESP_LOGE(TAG, "POST failed: %s", esp_err_to_name(err));
    }

    esp_http_client_cleanup(client);
    cJSON_free(json);
    cJSON_Delete(root);

    return err;
}
```

Додаток Б

Сайт

```jsx
import { useState } from "react";
import ControlsPanel from "../components/ControlsPanel.jsx";
import ChartCard from "../components/ChartCard.jsx";
import { fetchMicroclimateData } from "../api/microclimateApi.js";

function normalizeItem(item, index) {
  if (Array.isArray(item)) {
    return {
      label: String(item[0] ?? `Інтервал ${index + 1}`),
      temperature: Number(item[1]),
      humidity: Number(item[2]),
      pressure_hpa: Number(item[3]),
      gas_resistance_ohm: Number(item[4]),
    };
  }

  return {
    label:
      item.intervalStart ||
      item.localDateTime ||
      item.timestamp ||
      item.time ||
      item.time_bucket ||
      `Інтервал ${index + 1}`,

    temperature: Number(item.temperature),
    humidity: Number(item.humidity),
    pressure_hpa: Number(item.pressure_hpa),
    gas_resistance_ohm: Number(item.gas_resistance_ohm),
  };
}

export default function MicroclimatePage() {
  const [from, setFrom] = useState(null);
  const [to, setTo] = useState(null);

  const [intervalValue, setIntervalValue] = useState("20");
  const [intervalUnit, setIntervalUnit] = useState("seconds");

  const [mode, setMode] = useState("average");
  const [data, setData] = useState([]);

  const [loading, setLoading] = useState(false);
  const [error, setError] = useState("");

  async function loadData() {
    setLoading(true);
    setError("");

    try {
      const json = await fetchMicroclimateData({
        from,
        to,
        intervalValue,
        intervalUnit,
        mode,
      });

      console.log("Відповідь backend:", json);

      setData(json.map(normalizeItem));
    } catch (e) {
      setError(e.message);
      setData([]);
    } finally {
      setLoading(false);
    }
  }

  return (
    <main className="page">
      <h1>Панель мікроклімату BME680</h1>

      <ControlsPanel
        from={from}
        to={to}
        intervalValue={intervalValue}
        intervalUnit={intervalUnit}
        mode={mode}
        setFrom={setFrom}
        setTo={setTo}
        setIntervalValue={setIntervalValue}
        setIntervalUnit={setIntervalUnit}
        setMode={setMode}
        onSubmit={loadData}
        loading={loading}
      />

      {error && <div className="error">{error}</div>}

      <section className="dashboard-grid">
        <ChartCard title="Вологість" unit="%" data={data} field="humidity" />
        <ChartCard title="Температура" unit="°C" data={data} field="temperature" />
        <ChartCard title="Тиск" unit="hPa" data={data} field="pressure_hpa" />
        <ChartCard title="Газовий опір" unit="Ω" data={data} field="gas_resistance_ohm" />
      </section>
    </main>
  );
}
```

Додаток В

Панель керування сайтом

```jsx
import DatePicker from "react-datepicker";
import "react-datepicker/dist/react-datepicker.css";

export default function ControlsPanel({
  from,
  to,
  intervalValue,
  intervalUnit,
  mode,
  setFrom,
  setTo,
  setIntervalValue,
  setIntervalUnit,
  setMode,
  onSubmit,
  loading,
}) {
  return (
    <section className="controls">
      <label>
        Час від
        <DatePicker
          selected={from}
          onChange={(date) => setFrom(date)}
          showTimeInput
          timeInputLabel="Час:"
          dateFormat="yyyy-MM-dd HH:mm:ss"
          className="custom-input"
          placeholderText="Оберіть дату і час"
        />
      </label>

      <label>
        Час до
        <DatePicker
          selected={to}
          onChange={(date) => setTo(date)}
          showTimeInput
          timeInputLabel="Час:"
          dateFormat="yyyy-MM-dd HH:mm:ss"
          className="custom-input"
          placeholderText="Оберіть дату і час"
        />
      </label>

      <label>
        Значення інтервалу
        <input
          type="number"
          min="1"
          value={intervalValue}
          onChange={(e) => setIntervalValue(e.target.value)}
        />
      </label>

      <label>
        Одиниця інтервалу
        <select
          value={intervalUnit}
          onChange={(e) => setIntervalUnit(e.target.value)}
        >
          <option value="seconds">Секунди</option>
          <option value="minutes">Хвилини</option>
          <option value="hours">Години</option>
        </select>
      </label>

      <label>
        Тип значення
        <select value={mode} onChange={(e) => setMode(e.target.value)}>
          <option value="average">Середнє</option>
          <option value="min">Мінімальне</option>
          <option value="max">Максимальне</option>
        </select>
      </label>

      <button
        onClick={onSubmit}
        disabled={loading || !from || !to || !intervalValue}
      >
        {loading ? "Завантаження..." : "Показати параметри мікроклімату"}
      </button>
    </section>
  );
}
```

```jsx
import ReactECharts from "echarts-for-react";

export default function ChartCard({ title, unit, data, field }) {
  const labels = data.map((item) => item.label);

  const values = data.map((item) => {
    const value = Number(item[field]);
    return Number.isFinite(value) ? value : 0;
  });

  const option = {
    tooltip: {
      trigger: "axis",
      valueFormatter: (value) => `${value} ${unit}`,
    },
    grid: {
      left: 50,
      right: 20,
      top: 40,
      bottom: 70,
    },
    dataZoom: [
      {
        type: "inside",
      },
      {
        type: "slider",
        bottom: 20,
      },
    ],
    xAxis: {
      type: "category",
      data: labels,
      axisLabel: {
        rotate: labels.length > 8 ? 45 : 0,
      },
    },
    yAxis: {
      type: "value",
      name: unit,
      scale: true,
    },
    series: [
      {
        name: title,
        type: "bar",
        data: values,
        barMaxWidth: 45,
      },
    ],
  };

  return (
    <article className="chart-card">
      <h2>{title}</h2>

      {data.length === 0 ? (
        <div className="empty">Оберіть період і натисніть кнопку показу</div>
      ) : (
        <ReactECharts
          option={option}
          style={{ height: "360px", width: "100%" }}
          notMerge={true}
          lazyUpdate={true}
        />
      )}
    </article>
  );
}
```

Додаток Г

Код серверної частини

```java
package com.hamaza;

import java.util.TimeZone;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class EspAndSpring {
    public static void main(String[] args) {
        TimeZone.setDefault(TimeZone.getTimeZone("UTC"));
        SpringApplication.run(EspAndSpring.class, args);
    }
}
```

```java
package com.hamaza.web;

import com.hamaza.model.Bme680Reading;
import com.hamaza.repository.Bme680ReadingRepository;
import org.springframework.web.bind.annotation.*;
import com.hamaza.service.Bme680ReadingService;

import java.util.List;
import java.util.Map;
import java.time.LocalDateTime;

@CrossOrigin(origins = "http://localhost:5173")
@RestController
@RequestMapping("/api/bme680/microclimate")
public class Bme680ReadingController {

    private final Bme680ReadingRepository repository;
    private final Bme680ReadingService service;

    public Bme680ReadingController(Bme680ReadingRepository repository, Bme680ReadingService service) {
        this.repository = repository;
        this.service = service;
    }

    @PostMapping
    public Bme680Reading saveReading(@RequestBody Map<String, Object> jsonData) {
        Bme680Reading reading = new Bme680Reading(jsonData);
        return repository.save(reading);
    }

    @GetMapping
    public List<Bme680Reading> getAllReadings() {
        return repository.findAll();
    }

    @GetMapping("/{id}")
    public Bme680Reading getReadingById(@PathVariable Long id) {
        return repository.findById(id)
                .orElseThrow(() -> new RuntimeException("Reading not found"));
```

```java
    @GetMapping("/max")
    public List<Object[]> getMaxByInterval(
        @RequestParam LocalDateTime from,
        @RequestParam LocalDateTime to,
        @RequestParam String interval) {
        return service.getMaxByInterval(from, to, interval);
    }

    @GetMapping("/min")
    public List<Object[]> getMinByInterval(
        @RequestParam LocalDateTime from,
        @RequestParam LocalDateTime to,
        @RequestParam String interval) {
        return service.getMinByInterval(from, to, interval);
    }

    @GetMapping("/average")
    public List<Object[]> getAverageReadings(
            @RequestParam LocalDateTime from,
            @RequestParam LocalDateTime to,
            @RequestParam String interval) {
        return service.getAverageReadings(from, to, interval);
    }
}
```

```java
package com.hamaza.repository;

import com.hamaza.model.Bme680Reading;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;
import org.springframework.stereotype.Repository;

import java.time.LocalDateTime;
import java.util.List;

public interface Bme680ReadingRepository extends JpaRepository<Bme680Reading, Long> {
    @Query(value = """
    SELECT
        date_bin(CAST(:interval AS interval), created_at, TIMESTAMP '1970-01-01') AS time_bucket,
        AVG((data->>'temperature')::numeric) AS temperature,
        AVG((data->>'humidity')::numeric) AS humidity,
        AVG((data->>'pressure_hpa')::numeric) AS pressure_hpa,
        AVG((data->>'gas_resistance_ohm')::numeric) AS gas_resistance_ohm
    FROM bme680_readings
    WHERE created_at BETWEEN :from AND :to
    GROUP BY time_bucket
    ORDER BY time_bucket
    """, nativeQuery = true)
    List<Object[]> findAverageReadings(
        @Param("from") LocalDateTime from,
        @Param("to") LocalDateTime to,
        @Param("interval") String interval);

    @Query(value = """
    SELECT
        date_bin(CAST(:interval AS interval), created_at, TIMESTAMP '1970-01-01') AS time_bucket,
        MAX((data->>'temperature')::numeric) AS max_temperature,
        MAX((data->>'humidity')::numeric) AS humidity,
        MAX((data->>'pressure_hpa')::numeric) AS pressure_hpa,
        MAX((data->>'gas_resistance_ohm')::numeric) AS gas_resistance_ohm
    FROM bme680_readings
    WHERE created_at BETWEEN :from AND :to
    GROUP BY time_bucket
    ORDER BY time_bucket
    """, nativeQuery = true)
    List<Object[]> findMaxByInterval(
        @Param("from") LocalDateTime from,
        @Param("to") LocalDateTime to,
        @Param("interval") String interval);

    @Query(value = """
    SELECT
        date_bin(CAST(:interval AS interval), created_at, TIMESTAMP '1970-01-01') AS time_bucket,
        MIN((data->>'temperature')::numeric) AS min_temperature,
        MIN((data->>'humidity')::numeric) AS humidity,
        MIN((data->>'pressure_hpa')::numeric) AS pressure_hpa,
        MIN((data->>'gas_resistance_ohm')::numeric) AS gas_resistance_ohm
    FROM bme680_readings
    WHERE created_at BETWEEN :from AND :to
    GROUP BY time_bucket
    ORDER BY time_bucket
    """, nativeQuery = true)
    List<Object[]> findMinByInterval(
        @Param("from") LocalDateTime from,
        @Param("to") LocalDateTime to,
        @Param("interval") String interval);
}
```

```java
package com.hamaza.service;
import com.hamaza.repository.Bme680ReadingRepository;
import org.springframework.stereotype.Service;

import java.util.List;
import java.time.LocalDateTime;

@Service
public class Bme680ReadingService {

    private final Bme680ReadingRepository repository;

    public Bme680ReadingService(Bme680ReadingRepository repository) {
        this.repository = repository;
    }

    public List<Object[]> getAverageReadings(
            LocalDateTime from,
            LocalDateTime to,
            String interval) {
        if (from.isAfter(to)) {
            throw new IllegalArgumentException("from must be before to");
        }

        return repository.findAverageReadings(from, to, interval);
    }

    public List<Object[]> getMaxByInterval(
        LocalDateTime from,
        LocalDateTime to,
        String interval) {
        if (from.isAfter(to)) {
            throw new IllegalArgumentException("from must be before to");
        }

        return repository.findMaxByInterval(from, to, interval);
    }

    public List<Object[]> getMinByInterval(
            LocalDateTime from,
            LocalDateTime to,
            String interval) {
            if (from.isAfter(to)) {
                throw new IllegalArgumentException("from must be before to");
            }

            return repository.findMinByInterval(from, to, interval);
    }
}
```